Actes des Cinquièmes journées nationales du Groupement De Recherche CNRS du Génie de la Programmation et du Logiciel

Laurence Duchien

Actes des Cinquièmes journées nationales du

Groupement De Recherche CNRS du Génie de la Programmation et du Logiciel

Université de Lorraine

Laboratoire LORIA

Inria Nancy - Grand Est

3 au 5 avril 2013

Editeur : Laurence Duchien

Impression : service de reprographie, Université de Lorraine

Table des matières

Préface

C'est avec grand plaisir que je vous accueille pour les Cinquièmes Journées Nationales du GDR GPL dans la belle ville de Nancy. Ces journées sont l'occasion de rassembler la communauté du GDR Génie de la Programmation et du Logiciel (GPL). Les missions principales du GDR GPL sont l'animation scientifique de la communauté et la promotion de nos disciplines, notamment en direction des jeunes chercheurs. Cette animation scientifique est d'abord le fruit des efforts de nos groupes de travail, actions transverses et de l'Ecole des Jeunes Chercheurs en Programmation.

Le GDR GPL est maintenant dans sa sixième année d'activité. Ces dernières années, les journées nationales se sont affirmées comme un temps fort de l'activité de notre GDR, l'occasion pour toute la communauté d'échanger et de s'enrichir des derniers travaux présentés. Plusieurs événements scientifiques sont co-localisés avec ces journées nationales : la 2nde édition de la conférence CIEL 2013, fusion des journées IDM et de la conférence LMO, ainsi que 12ème édition d'AFADL 2013, atelier francophone sur les Approches Formelles dans l'Assistance au Développement de Logiciels.

Ces journées sont une vitrine où chaque groupe de travail ou action transverse donne un aperçu de ses recherches. Une trentaine de présentations ont ainsi été sélectionnées par les responsables des groupes de travail. Comme les années précédentes, nous avons demandé aux groupes de travail de nous proposer, en règle générale, des présentations qui avaient déjà fait l'objet d'une sélection dans une conférence nationale ou internationale ; ceci nous garantit la qualité du programme.

Trois conférenciers invités nous ont fait l'honneur d'accepter notre invitation. Il s'agit de Yves Le Traon (Université du Luxembourg et FSTC/SnT) dont la présentation sera commune à la conférence CIEL, de Laurent Voisin (Société Systerel) dont la présentation sera commune à AFADL, et de Bruno Legeard (Femto-st et Smartesting). Une table ronde, animée par Jean-Marc Menaud et Romain Rouvoy, abordera l'économie d'énergie dans les infrastructures et les logiciels.

Le GDR GPL a à cœur de mettre à l'honneur les jeunes chercheurs. C'est pourquoi nous avons décidé de créer un prix de thèse du GDR. Nous aurons le plaisir de remettre ce premier prix de thèse en Génie du Logiciel et de la Programmation à Gabriel Keirneis pour sa thèse intitulée *Continuation-Passing C : Transformations de programmes pour compiler la concurrence dans un langage impératif.* Le jury chargé de sélectionner le lauréat a été présidé par Dominique Méry. Que ce dernier soit ici remercié ainsi que l'ensemble des membres du jury, pour tout le travail accompli.

Ces journées ont aussi pour objectif de préparer l'avenir en favorisant l'intégration des jeunes chercheurs dans la communauté et leur future mobilité. Dans cet esprit, nous les avons encouragés à proposer un poster ou une démonstration de leurs travaux, en leur offrant les frais d'inscription. Une vingtaine ont répondu à cet appel. Un prix du meilleur poster sera égalemment remis pendant ces journées par un jury présidé par Jean-Louis Giavitto.

Avant de clôturer cette préface, je tiens à remercier tous ceux qui ont contribué à l'organisation de ces journées nationales : les responsables de groupes de travail ou d'actions transverses, les membres du comité de direction du GDR GPL et, tout particulièrement, le comité d'organisation de ces journées nationales présidé par Pierre-Etienne Moreau. Je remercie chaleureusement l'ensemble des collègues nancéiens qui n'ont pas ménagé leurs efforts pour nous accueillir dans les meilleures conditions.

Laurence Duchien
Directrice du GDR Génie de la Programmation et du Logiciel

Comités

Comité de programme des journées nationales

Le comité de programme des journées nationales 2013 est composé par les membres du comité de direction du GDR GPL et les responsables de groupes de travail.

Laurence Duchien (présidente), LIFL, Université Lille 1

Yamine Ait Ameur,LISI / ENSMA
Mireille Blay-Fornarino, I3S, Université Nice-Sophia-Antipolis
Yohan Boichut, LIFO, Université d'Orléans
Isabelle Borne, IRISA, Université de Bretagne Sud
Frédérice Dadeau, FEMTO-ST, Université de Franche-Comté
Catherine Dubois, CEDRIC, ENSIIE
Lydie Du Bousquet, LIG, Université Joseph Fourier
Frédéric Gava, LACL, Université Paris-Est
Jean-Louis Giavitto (Président du jury des posters), IRCAM, CNRS
Laure Gonnord, LIFL, Université Lille 1
Gatan Hains, LACL, Université Paris-Est
Pierre-Cyrille Heam, FEMTO-ST, Université Franche-Comté
Akram Idani, LIG, ENSIMAG
Claude Jard, IRISA, ENS-Cachan en Bretagne
Thomas Jensen, IRISA, CNRS
Yves Ledru, LIG, Université Joseph Fourier
Pierre-Etienne Moreau, LORIA, Inria
Pascal Poizat, LIP6, Université Paris-Nanterre
Marc Pouzet, Ecole Normale Supérieure, Université Pierre et Marie Curie, IUF
Fabrice Rastello, Inria, ENS Lyon
Romain Rouvoy, LIFL, Université Lille 1
Olivier H. Roux, IRCCyN, Université de Nantes
Salah Sadou, VALORIA, Université Bretagne-Sud
Christel Seguin, ONERA Centre de Toulouse
Chouki Tibermacine, LIRMM, Université Montpellier II
Sarah Tucci, CEA LIST

Comité scientifique du GDR GPL

Franck Barbier(LIUPPA, Pau)
Charles Consel (LABRI, Bordeaux)
Roberto Di Cosmo (PPS, Paris VII)
Christophe Dony (LIRMM, Montpellier)
Stéphane Ducasse (Inria, Lille)
Jacky Estublier (LIG, Grenoble)
Nicolas Halbwachs (Verimag, Grenoble)
Marie-Claude Gaudel (LRI, Orsay)
Gatan Hains (LACL, Créteil)
Valérie Issarny (Inria, Rocquencourt)
Jean-Marc Jézéquel (IRISA, Rennes)
Dominique Méry (LORIA, Nancy)
Christine Paulin (LRI, Orsay)

Comité d'organisation

Pierre-Etienne Moreau (président), Université de Lorraine
Jean-Christophe Bach, Inria Nancy Grand Est
Laurence Benini, Inria Nancy Grand Est
Christophe Calves, Université de Lorraine
Anne-Lise Charbonnier, Inria Nancy Grand Est
Horatiu Cirstea, Université de Lorraine
Jean-Pierre Jacquot, Université de Lorraine
Serguei Lenglet, Université de Lorraine
Louisa Touioui, Inria Nancy Grand Est
Olivier Zendra, Inria Nancy Grand Est

Conférenciers invités

Security testing : A key challenge for software engineering

Auteur : Yves Le Traon (Université du Luxembourg, FSTC/SnT)

Résumé :
While important efforts are dedicated to system functional testing, very few works study how to specifically and systematically test security mechanisms. In this talk, we will present two categories of approaches. The first ones aim at assessing security mechanisms compliance with declared policies. Any security policy is strongly connected to system functionality : testing function includes exercising many security mechanisms. However, testing functionality does not intend at exercizing all security mechanisms. We thus propose test selection criteria to produce tests from a security policy. Empirical results will be presented about access control policies and about Android apps permission checks. The second ones concern the attack surface of web apps, with a particular focus on web browser sensitivity to XSS attacks. Indeed, one of the major threats against web applications is Cross-Site Scripting (XSS) that crosses several web components : web server, security components and finally the client's web browser. The final target is thus the client running a particular web browser. During this last decade, several competing web browsers (IE, Netscape, Chrome, Firefox) have been upgraded to add new features for the final users benefit. However, the improvement of web browsers is not related with systematic security regression testing. Beginning with an analysis of their current exposure degree to XSS, we extend the empirical study to a decade of most popular web browser versions.The results reveal a chaotic behavior in the evolution of most web browsers attack surface over time. This particularly shows an urgent need for regression testing strategies to ensure that security is not sacrificed when a new version is delivered.
In both cases, security must become a specific target for testing in order to get a satisfying level of confidence in security mechanisms

Validation Formelle de Données

Auteur : Laurent Voisin (Société SYSTEREL)

Résumé :
Dans le domaine ferroviaire, il est d'usage de développer un logiciel générique d'une part et des données de configuration d'autre part. Ces données sont élaborées par une équipe indépendante. Les données contiennent une description physique de l'environnement (par ex. la géométrie d'un réseau ferré) ainsi que des informations d'origine technique (par ex. un plan d'adressage réseau).
Se pose alors la question de la validité des données de configuration (représentent-elles bien la réalité du terrain ?), de leur cohérence et de leur conformité aux attendus du logiciel qui les reçoit. Classiquement, ces points sont vérifiés par relecture manuelle ou automatisés par l'utilisation de logiciels de vérification dédiés.
Une nouvelle approche de validation a fait son apparition ces dernières années et a été implantée dans l'outil OVADO développé par la RATP. Il s'agit de décrire les attendus sur les données sous la forme de propriétés mathématiques, puis d'évaluer ces propriétés sur un jeu de données pour s'assurer que ce dernier est conforme. Cette nouvelle approche apporte les avantages suivants par rapport aux approches plus classiques : la validation est facile à répéter (contrairement à une lecture manuelle), elle profite de toute la rigueur des méthodes formelles et enfin elle reste très souple : en cas de changement des exigences, il est plus simple de changer une spécification sous forme de propriété mathématique qu'une implantation sous forme de logiciel de vérification.

Biographie :
Laurent Voisin est le responsable R&D de la société SYSTEREL. De formation ingénieur, il a commencé sa carrière dans le domaine du génie logiciel (compilation, atelier de génie logiciel, exécutifs temps-réel), puis s'est tourné vers les méthodes formelles, en particulier la méthode B. Il a été le responsable du développement et de la maintenance de l'Atelier B pour la société CLEARSY de 1999 à 2004. Il a ensuite secondé Jean-Raymond Abrial à l'ETH Zurich de 2004 à 2007, où il a défini l'architecture et mené le développement de la plate-forme RODIN. Depuis 2007, il a rejoint la société SYSTEREL où il a continué de mettre en oeuvre les méthodes formelles dans un contexte industriel.

Génération de tests à partir de modèle. Retour sur 10 ans d'expérience de transfert de technologie

Auteur : Bruno Legeard (Femto-st & Smartesting)

Résumé :

De BZ-Testing-Tools au début des années 2000 à Smartesting qui fêtera ses 10 années d'existence fin 2013, cette présentation décrit une histoire de transfert de technologie dans le domaine de la génération de tests à partir de modèle. Quelles ont été les grandes étapes du passage du labo au marché ? Quel positionnement sur le marché des outils du test logiciel ? Quels enseignements peuvent servir aux chercheurs en Génie Logiciel qui seraient tentés aujourd'hui par une telle démarche ? Ce sont quelques-unes des questions qui seront abordées durant l'exposé.

Biographie :
Bruno Legeard est Professeur à l'Université de Franche-Comté (Institut Femto-st) et co-fondateur de l'entreprise Smartesting. Il travaille depuis la fin des années 90 sur la génération de tests à partir de modèles fondés sur des techniques symboliques. Les technologies développées sont utilisées dans le domaine de l'IT (grands systèmes d'information) et sur certaines parties de l'embarqué (carte à puce, paiement). Bruno est coauteur des livres "Practical Model-Based Testing" Morgan & Kaufmann 2006 et "Industrialiser le test fonctionnel" DUNOD 2009 & 2011.

Sessions des groupes de travail

Session de l'action AFSEC

Approches Formelles des Systèmes Embarqués Communicants

Markov Decision Process for Energy Management in Wireless Sensor Networks

Benoît Barbot[1], Marco Beccuti[2], G. Franceschinis[3] and Serge Haddad[1]

[1] LSV, ENS Cachan, CNRS & INRIA; France
[2] Dipartimento di Informatica, University of Torino; Italy
[3] Dipartimento di Informatica, Università del Piemonte Orientale; Italy

Abstract. We address the problem of a sensor network whose goal is to detect some attackers inside a critical area while consuming as little energy as possible. We propose a modelling of this problem by a partially observed Markov decision process (POMDP). While generic POMDP solvers fail to handle this model even for small parameter values, we design a dedicated method based on a statistical model-checker that is able to handle larger networks while providing an accurate strategy for the sensors.

1 Introduction

In the last years, the rapid progress in sensor technology and wireless communication has made Wireless Sensor Networks (WSNs) possible. They are composed by several sensor nodes, each with the ability of acquiring information from the surrounding environment, of processing them locally and of transferring them to a collecting node (also called sink) by means of a wireless connection. The nodes are usually powered by a battery and use a multi-hop approach to transfer their data to the sink node; their organization can be completely distributed but can also be coordinated by one or more powerful nodes that are not battery operated (e.g. the sink node). The information collected though the sink is then processed by a back-end system to obtain a global view of the system and take decisions.

The low cost and the wireless communication capability of WSN make very appealing their employment in several monitoring applications as environmental monitoring, military surveillance, and home and industrial security. However, in most of these applications, the power consumption of each node is a critical aspect which can impose severe limitations on energy resources, computational power and reliability of communication making the network subject to malfunctions and unavailability.

In this paper, we address the problem of power consumption in WSN when the most energy consuming part of the nodes are their sensors. In this setting, computing efficient strategies to decide if the sensor of a specific node should be on or off is the best way to reduce energy consumption. For each node, this strategy will be based on the observation of other sensors. As all the sensors

are not always switched on, the observation of the system is partial, leading to model the system as Partially Observed Markov Decision Process (POMDP).

Finding optimal strategies for POMDP is not an easy task, the solution can be (multiply) exponential with respect to the number of sensors and to the time horizon. The problem is even undecidable for infinite horizon. To circumvent this limitation, we present approximated solutions built iteratively using simulations. Applying these methods on examples yields solutions close to optimal on small models with finite time horizon.

2 Modelling the problem

The problem we are addressing is the following one. A zone is watched by a team of sensors, each one controlling a part of the zone called an area. From time to time, a drone may appear in the zone and must be detected and tracked by the sensors. When it is inside the zone the drone may randomly move inside from areas to contiguous ones or exit. We represent this zone by a graph $G = (V_\perp, E)$ where the set of vertices $V_\perp = V \cup \{\perp\}$ is decomposed in vertices of V, associated with the areas and a virtual position $\perp$ representing the outside of the zone. An oriented edge $e \in E$ between two vertices indicate that within one time unit, the drone can move from the source area to the destination area. The presence of an arc and its probability may depend on geographical characteristics of the areas composing the controlled zone, and in practice it may possibly be derived from previous observations of the drone movements. We allow loops in the graph with the meaning that the drone can stay in the same area. So the behaviour of the drone is specified by a transition matrix $\mathbf{P}_m$ with:

$$\forall v \in V_\perp \quad \sum_{(v,v') \in E} \mathbf{P}_m(v, v') = 1$$

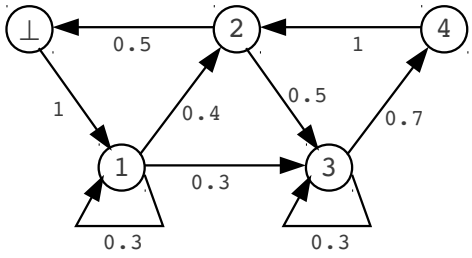


Fig. 1. Modelling the zone

For instance in the zone modelled by the graph of figure 1, the drone surely enters the zone in area 1. When it is in area 1, it can stay there with probability

0.3 or leave it moving to area 2 (resp. area 3) with probability 0.4 (resp. 0.3), etc.

Every sensor may be active or passive in order to save energy. Thus the actions that a sensor can perform at some instant is to keep its current mode or to switch it. Moreover a sensor detects the presence of the drone in its area if and only if it is active. We have represented on the left of figure 2, a situation with the drone located in area 3. Sensors 2 and 4 are active while sensors 1 and 3 are inactive. So the drone is not detected. On the right part of the figure, we have represented the state graph of any sensor.

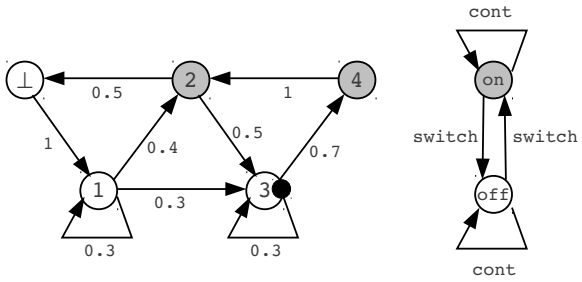


Fig. 2. Watching the zone looking for a drone

The reward of the system is based on the states of the sensors.

- There is a cost (i.e. a negative reward) r_{on} for a sensor to be active.
- There is a reward r_{det} when a sensor is active and the drone is located in the corresponding area.

Observe that the value of r_{on} can be derived from the battery consumption due to the sensor being ON, so that it can be derived in an objective way from the sensor specification, while r_{det} depends on how critical it is to miss the drone when it is over a given area.

The decision of every sensor is based on the part of history it can be aware of and this part is common to all sensors. Indeed we assume that all sensors are connected to some central station and that communications are "instantaneous": local information can be transmitted to the station and then broadcast to every sensor at each time step. Observe that these assumptions allow to obtain a relatively simple model and can be explained as follows: the "instantaneous" update of the global system state information at each sensor is reasonable if an appropriate length for the time step is chosen (consistently with the hypothesis of the drone being able to move between two adjacent areas in one time step). The ability to directly communicate between the sensors and the central station may be realistic along the way from the central station to the sensors (broadcast of the system state), while in the opposite direction a multi-hop delivery mechanism might be needed.

The global system state known by every sensor in the net, consists of:

– the states of all sensors from the initial state up to the current state,
– the different positions of the drone that have been detected by the active sensors when the drone was in their local area.

3 Solving the problem

Let us observe that whatever the global reward one wants to maximize, the computation is highly untractable and in some cases impossible. Indeed:

– The number of states is already exponential w.r.t. the number of sensors.
– Since the model is a POMDP, the decision rules consist of piecewise linear and convex mappings with a number of linear mappings that may grow exponentially with time horizon.
– In the context of infinite horizon, there is no hope for a state-based strategy since when the drone is undetected, the strategy must take into account the last detected position but also the time elapsed since this detection.

We propose an iterative approximative method that works as follows.

We restrict the search for strategy among the one based only on the last known position of the drone and on the current detection status. Those strategies can be encoded as an automaton with $2n$ states, with n being the number of sensors. Each of these states specifies which sensors are switched on.

At each iteration, we simulate the synchronized product of the Markov chain describing the movement of the drone with the automaton that implements the strategy. During this simulation we compute an estimation of the probability to move from states of the automaton to others. Those transition probabilities are stored in a $4n^2$ matrix denoted P_{last}. From this matrix we can deduce a new strategy for the next iteration.

The computation of the new strategy is done by transforming the matrix P_{last} into a set of Markov Decision Process (MDP) corresponding to the point of view of a single node of the WSN, assuming that all the behaviors of the drone and of the others sensors is Markovian and following transition probability describe by P_{last}. Finally the new strategy combines the optimal strategy of each sensors computed by solving the MDP.

Experiments show us that these iterative methods can lead to solutions with a reward close to the one we can obtained by solving the POMDP when such a solution can be computed. When the number of sensors does not allow the use of analytic tools, our method still allows to compute an efficient strategy.

4 Conclusion

Optimization of the energy consumption in wireless sensor network can be modeled as POMDP. Analytic solutions of such POMDP are often intractable. We provide here an iterative method based on simulation which allows to compute an efficient policy to solve this problem.

Ingénierie formelle pour logiciels aéronautiques certifiés

Frédéric Boniol et Virginie Wiels

ONERA, Centre de Toulouse,
2, Avenue E. Belin, BP 74025, 31077 Toulouse Cedex 4, France
{Frederic.Boniol,Virginie.Wiels}@onera.fr

Abstract. Le DO-178 est le standard de certification pour les logiciels aéronautiques, il définit des objectifs de vérification pour chaque étape de développement du logiciel (exigences, conception, code source, code exécutable). Nous concevons à l'Onera une chaîne de développement et d'analyses supportée par des langages et des techniques de vérification formels. Nous présenterons les différents outils de cette chaîne et comment ils permettent de répondre aux objectifs du DO-178. Cette chaîne de développement sera expérimentée sur la production d'un code de contrôle-commande pour un drone avion.

Runtime Verification for Real-Time Automotive Embedded Software

Sylvain Cotard*∓, Sébastien Faucou∓, Jean-Luc Béchennec⋆,
Audrey Queudet∓, and Yvon Trinquet∓

Renault SAS*, LUNAM Université. Université de Nantes∓, CNRS⋆
IRCCyN UMR CNRS 6597
44321 Nantes, France
firstname.name@irccyn.ec-nantes.fr

Abstract. We present the design of an error detection service for real-time automotive embedded software. The service monitors at runtime the data flows in a graph of communicating real-time tasks. At design-time, monitors are automatically generated from formal models; at compile-time, monitors are embedded in the Real Time Operating System (RTOS) kernel; at runtime, errors are detected and notified with a small and deterministic latency.

Keywords: embedded software, error detection, runtime verification, RTOS, AUTOSAR

1 Introduction

During the past 15 years, the number of services provided in vehicles caused the evolution of Electrics and Electronics (E/E) systems from federated architectures (one function per Electronic Control Unit (ECU)) to integrated architectures (several functions per ECU). In this context, automotive OEMs (Original Equipment Manufacturers) and suppliers are turning toward real-time and multitask-capable operating systems for improved code quality and efficiency. To face new challenges induced by theses changes, automotive industry stakeholders are working on the design of a common architecture supported by standardized software services: AUTOSAR (AUTomotive Open System ARchitecture) [1].

The context of our work is the dependable design of AUTOSAR systems. Among the attributes of dependability, we focus here on software fault tolerance and more specifically on error detection. We are developing an error detection service based on runtime verification in AUTOSAR-like systems [5].

The paper is organized as follows. In section 2 we present the main motivations of our work. In section 3, we recall the object of runtime verification and expose the runtime verification technique of LTL formula from bauer2011. In section 4, we explain how to use runtime verification efficiently in the context of automotive embedded systems. In section 5, we conclude.

2 Motivations

Modern automotive embedded software applications are composed of communicating real-time tasks. Their global behavior depends on many design time and runtime parameters. As an illustration, let us consider the model described in Figure 1. In this example, three concurrent tasks T_0, T_1 and T_2 communicate through two shared buffers b_0 and b_1. T_2 reads data from both b_0 and b_1 to make a coherency check between the input and the output of T_1. A correctness requirement for this application could be: *when T_2 starts reading, the buffers are synchronized and stay synchronized until it has finished.* The buffers are synchronized if the data currently stored in b_1 has been produced with the data currently stored in b_0.

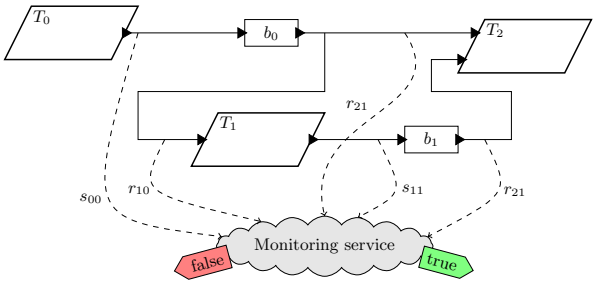


Fig. 1. Monitoring architecture. s_{xy} are sending events by T_x to b_y and r_{xy} are receiving events by T_x from b_y.

The satisfaction of such a property depends on parameters such as task scheduling parameters (period, priority, execution times), synchronisation and communication protocols, task assignment to cores (for multicore architecture), etc.

In an integrated architecture, several concurrent real-time applications are hosted on a single execution platform. Today, these platforms are based on monocore microcontrollers. In the near future, multicore microcontrollers (i.e. the combination of two or more calculation units on a same die, that run in parallel) will be used because they offer undeniable advantages in terms of performance and power consumption. Unfortunately, real-time parallel programming is known to be very difficult.

It is thus reasonable to consider that errors will occur at runtime in tomorrow's automotive embedded software. Then, runtime mechanisms to detect and mitigate these errors must be proposed.

A well known solution is to rely on diversification. Our proposal is a specific form of diversification, targeting data flow errors in real-time multitask software systems. Here, we focus on the error detection part. At design time, the expected

behaviors of the data flows are specified. They are used with a formal model of the system to generate specialized monitors. The tool *Enforcer*[1] has been built for this purpose. Then, at compile-time, the generated monitors are embedded in the RTOS kernel. Lastly, at runtime, the error detection service uses the monitors to report the behaviors that do not conform to the specification.

3 Runtime Verification

3.1 From Model Checking to Runtime Verification

Runtime Verification (RV) is a lightweight formal method that shares some concepts with *Model Checking* (MC). Both methods ask the designer to specify the properties ϕ that the system should verify. These properties are typically expressed with a temporal logic such as LTL (for RV or MC) or CTL (for MC). In MC, the designer must also provide a model M of the system, typically in the form of a transition system. Then, the model checker solves the problem $M \models \phi$. The answer is either yes, or a counter-example. MC allows to detect design errors.

In RV, the property ϕ is used to generate an event-based monitor that is translated into code to decide at runtime $\sigma \models \phi$, where σ denotes the ongoing execution of the system. The designer must provide some extra information to recognize and preprocess the events of interest. When the monitor receives an event, it outputs a verdict: $true$ (all the possible continuations of the execution will be accepted), $false$ (none of the continuations of the execution will be accepted), or $inconclusive$ (some continuations will be accepted, some others will not). The monitor must be built such that it outputs its verdict as soon as possible. Runtime verification allows to detect errors that are activated at runtime. That runtime verification can be introduced in industrial real-time embedded systems with a minimal execution time overhead and an acceptable memory footprint as shown in [4].

3.2 Runtime Verification of LTL formulae

Linear Temporal Logic (LTL) Temporal logics are mathematical tools that deal with the temporal behaviors of discrete event systems. LTL (Linear Temporal Logic) is a temporal logic proposed by Pnueli for the formal specification and verification of reactive systems [8]. LTL formulae express properties about the running of such systems. LTL extends propositional logic with two modalities: X (for neXt) and U (for Until), presented below.

Syntax: Let AP be a set of atomic propositions. The set of LTL formulae over AP is defined inductively as follows: if $p \in AP$, then p is a LTL formula; if ϕ and ψ are LTL formulae then $\neg\phi$, $\phi \wedge \psi$, $\phi U \psi$ and $X\phi$ are also LTL formulae.

[1] *Enforcer* is developed by the group *systèmes temps réel* at IRCCyN. It is distributed under GPL licence. It is available here: `http://enforcer.rts-software.org`

Semantics: Let $\Sigma = 2^{AP}$ and $\sigma = s_0 s_1 s_2 \dots s_i \dots \in \Sigma^\omega$ an ω-word on Σ. Let $\sigma(i) = s_i$ the i^{th} element of σ, and $\sigma_i = s_i s_{i+1}...$ the suffix of σ starting at the i^{th} element. Let $p \in AP$ and ϕ and ψ two LTL formulae over AP. The satisfaction relation $\sigma \models \phi$ is defined inductively as follows: $\sigma \models true$; $\sigma \models p$ iff $p \in \sigma(0)$; $\sigma \models \neg\phi$ iff $\sigma \not\models \phi$; $\sigma \models \phi \wedge \psi$ iff $\sigma \models \phi$ and $\sigma \models \psi$; $\sigma \models X\phi$ iff $\sigma_1 \models \phi$ (ϕ will be true at the next step); $\sigma \models \phi U \psi$ iff $\exists j \in \mathbb{N}$ s.t $\sigma_j \models \psi$ and $\forall k < j, \sigma_k \models \phi$ (ϕ remains true until ψ becomes true).

From these basic operators, it is possible to define other logical operators ($\vee$, $\Rightarrow$, ...) and modalities such as F (a property will eventually be true) and G (a property is and will always be true).

Automatic Generation of Monitors We recall here the construction of an RV monitor for the LTL formulae proposed in [2]

The monitor is given in the form of a Moore machine. The input alphabet of the machine is the set 2^{AP} where AP is the set of atomic propositions used to write ϕ. The output alphabet is the set $\mathbb{B}_3 = \{\top, \bot, ?\}$ (resp. true, false and inconclusive).

The procedure to build the machine is illustrated by figure 2. It is composed of two similar branches. The top branch builds a monitor that outputs either $\top$ or ? for formula ϕ. The bottom branch is the same for formula $\neg\phi$ so that the outputs of the monitor can be interpreted as $\bot$ and ?.

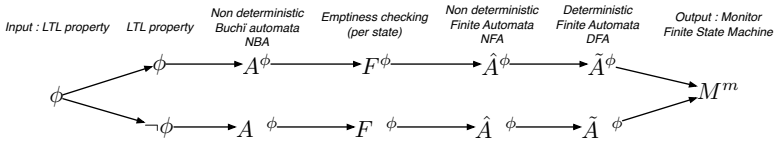


Fig. 2. Procedure to build a Moore machine from LTL formulae [2]

Consider the formula $\phi = G\ a$ (a is and will always be true). The first step consists in computing two non-deterministic Büchi automata (NBA) accepting the same $\omega-$languages as ϕ and $\neg\phi$. This can be done by using for instance the technique described by Gastin and Oddoux in [7]. The result is given on Table 1 (left side).

The next steps consist in computing two deterministic finite automata (DFA) that recognize the languages of the prefixes of the ω-word accepted by the two NBA. This requires to perform two classical operations: first, emptiness checking for each state of each NBA; then, determinization of finite automata. The result is given on Table 1 (middle). It is worth noting that a trap state has been added during the determinization step of ϕ so that the resulting automata is complete with regards to 2^{AP}.

The Moore machine is finally built by a synchronized product of the two DFA. The output function is then computed to associate a verdict $\top$, $\bot$ or ? to each state. The result is given on Table 1.

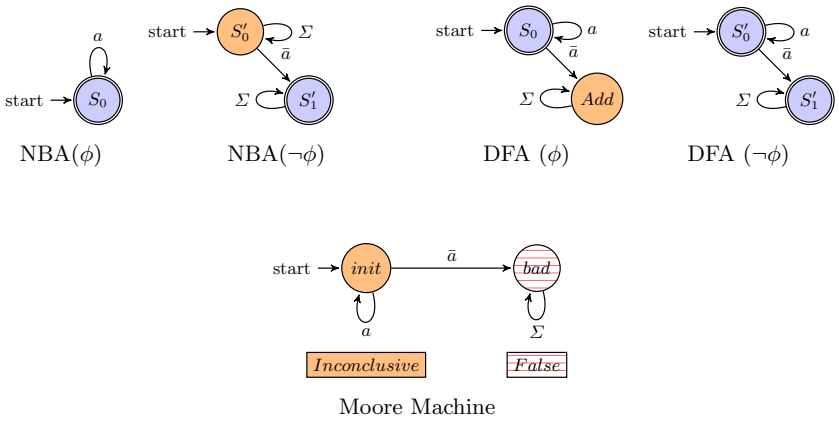


Table 1. Procedure for the Moore Machine synthesis (monitor for a given LTL formula)

RV of LTL formulae has some limits. Indeed, there exist LTL properties for which it is not possible to build a monitor that outputs $\top$ or $\bot$ in finite time. For these formulae, the procedure builds a Moore machine with a single state that outputs ?. Detailed discussion on the class of properties that can be monitored can be found in [6] and [2]. In [2], Bauer *et al.* describe an experiment based on a set of patterns commonly found in the specification of concurrent and reactive systems. On 107 patterns expressed as valid LTL formulae, 53 are found monitorable. This tends to show that, despite its limits, this technique can be used for a broad range of properties.

4 Towards an Error Detection Service based on Runtime Verification

RV appears to be a promising approach to generate monitors that could be used by an error detection service. This service should implement the following functions: identification of the event of interest ; update of the set of atomic properties that are true after the occurrence of an event ; computation of the output of the monitor to this update ; if the output is either $\top$ or $\bot$, notification to the error mitigation component. We have developed such a service (and the companion toolchain) for an AUTOSAR-like platform based on the Trampoline RTOS.

4.1 Implementation Constraints in Automotive Embedded Systems

A typical automotive embedded system is hosted on a microcontroller with limited resources (from 32KB to 1MB of RAM, a few MB of Flash, processor frequency under 100MHz) and must fulfill real-time constraints (required response time for some motor control functions is less than 1 ms). These constraints must be taken into account in the design of the error detection service. In other words, the service must accomplish its functions with a small and predictable overhead (both in time and memory) and it must also offer a predictable response time (ie. the time between the date of the occurrence of the event which allows to deduce that the property is verified/violated and the date of the notification of this deduction to the error mitigation component must be bounded).

As we are targeting the automotive domain, the characteristics of AUTOSAR must also be taken into account. One important characteristic is the static nature of AUTOSAR: all software objects are created at compile-time. A direct consequence is that all objects are known a priori. This allows to use specialization to achieve low and deterministic overhead for system services.

4.2 Efficient Identification and Preprocessing of the Events

To minimize the amount of runtime computation required to identify and preprocess the events, we have taken the following design decisions:

- the events can only be system calls. The set of system calls to intercept are identified offline. The identification consists in a triplet (called function, caller id, parameter values) ;
- the preprocessing of the events is also performed offline.

The first decision allows to automatically inject the event identification code in the source code of the RTOS kernel. The identification consists in a lookup in a table. This is done in $O(1)$.

To realize the second decision, we must bridge the gap between the intercepted events and the atomic proposition used to write the monitored properties. This is done with a deterministic finite automaton (DFA) that models the monitored system, and a labeling function that decorates the state of this DFA with atomic propositions.

Formally, the DFA A^s over alphabet Σ^s is defined as $A^s = (Q^s, i^s, \rightarrow_s)$ where Q^s is the finite set of states, $i^s \in Q^s$ is the initial state and $\rightarrow_s \subset (Q^s \times \Sigma^s) \mapsto Q^s$ is the transiton function.

The type of the labeling function is given by $\lambda^s \subset Q^s \mapsto 2^{AP}$.

The monitor computed by the RV technique is given by $M^m = (Q^m, i^m, \rightarrow_m, \gamma^m)$ where Q^m is the finite set of states, $i^m \in Q^m$ is the initial state, $\rightarrow_m \subset (Q^m \times 2^{AP}) \mapsto Q^m$ is the transition function and $\gamma^m \subset Q^m \mapsto \mathbb{B}_3$ is the output (injective) function.

To preprocess the events, we compute the Moore machine M' offline over Σ^s defined as $M' = (Q', i', \rightarrow, \gamma')$ where $Q' = Q^s \times Q^m$, $i' = (i^s, i^m)$, $\rightarrow \subset (Q' \times$

$\Sigma^s) \mapsto Q'$ where $(q^s, q^m) \xrightarrow{\sigma} (r^s, r^m)$ iff $q^s \xrightarrow{\sigma}_s r^s$ and $q^m \xrightarrow{u}_m r^m$ and $u \subseteq \lambda^s(r^s)$ and $\gamma^m(q^m) =?$, and $\gamma' \subset Q' \mapsto \mathbb{B}_3$ where $\gamma'(q^s, q^m) = \gamma^m(q^m)$.

Notice that we do not build the transitions outgoing from a state that outputs either $\bot$ or $\top$. When such a state is reached, the work of the monitor is finished. In practice we build only the subset of Q' composed of reachable states with a depth-first exploration starting at (i^s, i^m).

The machine M' reacts directly to the intercepted events. If the machine is encoded with a matrix, this reaction consists in another lookup in a table and can be done in $O(1)$.

The update of machine M' is performed after the identification step. Both steps being in $O(1)$, the time overhead is deterministic. We have performed experiments to confirm that this overhead is small enough (see [4]). The system is static, so the memory overhead can be estimated offline. In our experiments, we have also confirmed that the memory overhead is compatible with the constraints of automotive embedded systems.

Lastly, to ensure a small and predictable response time, all the steps are executed in kernel mode, ensuring freedom of interference from application tasks or interrupts.

4.3 Example

Let us consider a system composed of two tasks communicating through a blackboard. For this system, we want to monitor the property *a message written in the buffer is always read before being overwritten*. Let the atomic proposition a denote "the buffer does not contain a message that has not been read and that has been overwritten". Then we have $\phi = G\ a$.

To preprocess the event, we can use an abstract model of the system A^s over the alphabet $\Sigma^s = \{SendMessage, ReceiveMessage\}$ that counts the number of successive occurrences of $SendMessage$ in the set $\{0, 1, +\}$. The labelling function is $\{(0 \mapsto a), (1 \mapsto a), (+ \mapsto \bar{a})\}$.

The monitor generated for $G\ a$ has already been given in table 1. The machine M', resulting from the construction explained above, is given in figure 3.

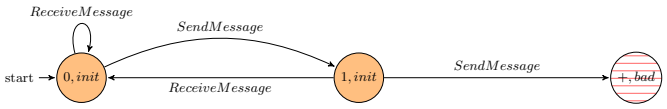


Fig. 3. Final monitor

4.4 Tool Support and Integration in Trampoline

We have developed a tool named *Enforcer*, that implements the generation of the machine M' [5]. *Enforcer* processes a model A^s and a property ϕ. It outputs the

source code (in C) of the machine M' that is statically injected in the kernel of the Trampoline RTOS [3]. The input language of *Enforcer* allows us to describe rules. Each rule contains a model of a part of the system and a property. The model is defined as a set of Deterministic Finite Automata (DFA) that are then composed with a synchronized product (A^S). The property is expressed in LTL over the set of state of the model.

The tool calls LTL2BA [2] to compute $NBA(\phi)$ and $NBA(\neg\phi)$ and implements all the other steps.

5 Conclusion

Our work aims at providing error detection and mitigation components for future real-time automotive embedded systems. We have designed a service for error detection. This service uses monitors that are automatically generated from formal models thanks to runtime verification techniques.

References

1. AUTOSAR. http://www.autosar.org. Technical report, AUTOSAR GbR, February 2012.
2. Andreas Bauer, Martin Leucker, and Christian Schallhart. Runtime verification for LTL and TLTL. *ACM Transactions on Software Engineering and Methodology*, 20(4), 2010.
3. Jean-Luc Béchennec, Mikaël Briday, Sébastien Faucou, and Yvon Trinquet. Trampoline - an open source implementation of the OSEK/VDX RTOS specification. In *IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 62–69, Prague, Tchèque, République, September 2006. IEEE.
4. Sylvain Cotard, Sébastien Faucou, and Jean-Luc Béchennec. A dataflow monitoring service based on runtime verification for. autosar os: Implementation and performances. In *Proceedings of the 8th annual workshop on Operating Systems Platforms for Proceedings of the 8th annual workshop on Operating Systems Platforms for Embedded Real-Time applications*, pages 46–55, 2012.
5. Sylvain Cotard, Sébastien Faucou, Jean-Luc Béchennec, Audrey Queudet, and Yvon Trinquet. A dataflow monitoring service based on runtime verification for. autosar. In *International Conference on Embedded Software and Systems (ICESS)*, 2012.
6. Yliès Falcone, Jean-Claude Fernandez, and Laurent Mounier. Runtime verification of safety-progress properties. In *International Workshop on Runtime Verification (RV)*, 2009.
7. Paul Gastin and Denis Oddoux. Fast LTL to Büchi automata translation. In *International Conference on Computer Aided Verification (CAV)*, pages 53–65, 2001.
8. Amir Pnueli. The temporal logic of programs. *Annual Symposium on Foundations of Computer Science (FOCS)*, pages 46–57, 1977.

[2] *LTL2BA* is developped by D. Oddoux and P. Gastin. It is available here: `http://www.lsv.ens-cachan.fr/gastin/ltl2ba` The output format of has been modified for our needs. The modified version is included in the *Enforcer* package.

Safety contracts for timed reactive components (extended abstract*)

Iulia Dragomir, Iulian Ober, and Christian Percebois

Université de Toulouse - IRIT
118 Route de Narbonne, 31062 Toulouse, France
{iulia.dragomir,iulian.ober,christian.percebois}@irit.fr

Abstract. A variety of system design and architecture description languages, such as SysML, UML or AADL, rely on the decomposition of complex system designs into communicating timed components. In this paper we consider the contract-based specification of such components. A contract is a pair formed of an assumption, which is an abstraction of the component's environment, and a guarantee, which is an abstraction of the component's behaviour given that the environment behaves according to the assumption. Thus, a contract concentrates on a specific aspect of the component's functionality and on a subset of its interface, which makes it relatively simpler to specify. Contracts may be used as an aid for hierarchical decomposition during design or for verification of properties of composites. This paper defines contracts for components formalized as a variant of timed input/output automata and introduces compositional results allowing to reason with contracts

1 Introduction

The development of safety critical real-time embedded systems is a complex and costly process, and the early validation of design models is of paramount importance for satisfying qualification requirements, reducing overall costs and increasing quality. Design models are validated using a variety of techniques, including design reviews [10], simulation and model-checking [4,11]. In all these activities system requirements play a central role; for this reason processes-oriented standards such as the DO-178C [7] emphasize the necessity to model requirements at various levels of abstraction and ensure their traceability from high-level down to detailed design and coding.

Since the vast majority of systems are designed with a component-based approach, the mapping of requirements is often difficult: a requirement is in general satisfied by the collaboration of a set of components and each component is involved in satisfying several requirements. A way to tackle this problem is to have partial and abstract component specifications which concentrate on specifying how a particular component collaborates in realizing a particular requirement; such a specification is called a *contract*. A contract is defined as a pair formed of

an *assumption*, which is an abstraction of the component's environment, and a *guarantee*, which is an abstraction of the component's behaviour given that the environment behaves according to the assumption.

The justification for using contracts is therefore manyfold: they support requirement specification and decomposition, mapping and tracing requirements to components and can be used in model reviews. Last but not least, contracts can support formal verification of properties through model-checking since, given the right composability properties, they can be used to restructure the verification of a property by splitting it in two steps: (1) verify that the components satisfy their corresponding contract and (2) the network of contracts correctly assembles and satisfies the property. Thus, one only needs to reason on abstractions when verifying a property, which potentially induces an important reduction of the combinatorial explosion problem.

Contracts have been introduced in object-oriented programming languages [8] and related concepts have since been defined for other component-based formalisms. Our contract framework is an instance of the generic framework proposed in [13,12], which formalizes the relations that come into play in such a framework and the properties that these relations have to satisfy in order to support reasoning with contracts.

Our interest in contracts is driven by potential applications in system engineering using SysML [9], in particular in the verification of complex industrial-scale designs for which we have reached the limits of our tools [3]. In SysML one can describe various types of communicating timed reactive components; for most of these, their semantics can be given in a variant of Timed Input/Output Automata (TIOA [5]). For this reason, this work concentrates on defining a contract framework for TIOA. The SysML layer, describing how contracts are defined and used in SysML, is left aside for space and complexity reasons and is subject to future work.

2 Contract-based Reasoning for Timed Reactive Systems

Being able to specify a contract for a component and to verify its satisfaction is an important step, but it is not sufficient to render the use of contracts interesting in a system design process. One also needs mechanisms for reasoning with contracts, i.e. check that the contracts for a set of components combine together to ensure the satisfaction of a global requirement on the composition of these components. Our framework follows a generic scheme for reasoning with contracts proposed by Quinton et. al. in [13,12].

In this scheme (illustrated in Figure 1), a system design is a hierarchical composition of components, these being Timed Input/Output Automata in our case. At each level of the hierarchy, n components $K_1, ..., K_n$ are combined to form a composite component $K_1 \parallel ... \parallel K_n$. The purpose of reasoning with contracts is to show that the composite satisfies a global property φ based on the contracts of $K_1, ..., K_n$, and avoiding the need to verify the property directly by model-checking the composite component, since this often leads to combinatorial

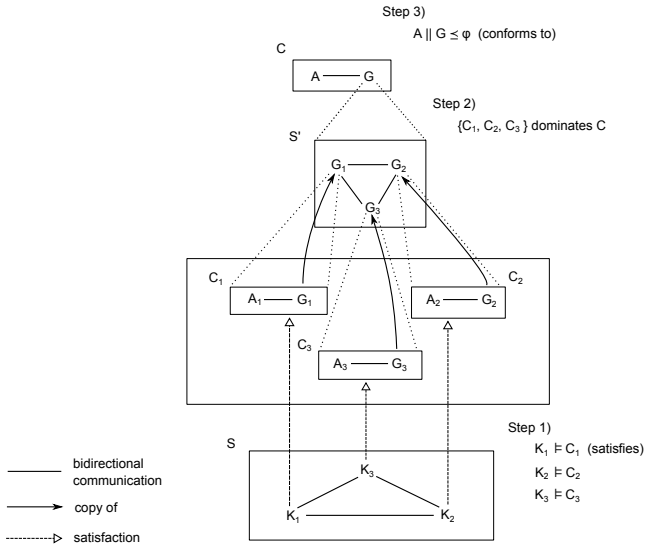


Fig. 1: Contract-based reasoning for a subsystem containing three components as presented in [12].

explosion. The contracts being specified by more abstract automata, one can assume that their composition will be less subject to explosion.

The reasoning proceeds as follows: for each component K_i, a contract C_i is given which consists of an abstraction A_i of the behaviour of K_i's environment, and an abstraction G_i that describes the expected behaviour of K_i given that the environment acts as A_i. Figure 1 presents three components K_1, K_2 and K_3 and a corresponding set of contracts C_1, respectively C_2 and C_3. Step 1 of the reasoning is to verify that each component is a correct implementation of the contract, i.e. the component *satisfies* its contract.

Step 2 of the reasoning consists in proving that the set of contracts $\{C_1, C_2, ..., C_n\}$ *imply* that the composite $K_1 \parallel ... \parallel K_n$ satisfies a contract $C = (A, G)$. To do so, [6] introduces a hierarchy relation between contracts, later called *dominance* in [12]: a set of contracts $\{C_1, C_2, ..., C_n\}$ *dominates* a contract C if and only if the composition of any valid implementations of $C_1, C_2, ..., C_n$ (hence, also $K_1 \parallel ... \parallel K_n$) is an implementation of C. As we will see later, to prove dominance one will have to verify certain conditions on compositions of assumptions and guarantees. In a multi-level hierarchy, the second step can be applied recursively up to a top-level contract (i.e. a contract for the whole system).

Finally, in the third step one has to prove that the system given by the top contract *conforms* to the specification φ (i.e. the property is satisfied): $A \parallel G \preceq \varphi$, where $\parallel$ denotes the usual parallel composition operator and $\preceq$ a *conformance* relation which in our case is TIOA language inclusion, φ also being specified as a TIOA.

The reasoning strategy presented here assumes that the system designer defines all the contracts necessary for verifying a particular requirement φ. How these contracts are produced is an interesting question but is outside the scope of this paper.

3 Timed Input/Output Automata

Many mathematical formalisms have been proposed in the literature for modelling communicating timed reactive components. We chose to build our framework based on a variant of Timed Input/Output Automata of [5] since it is one of the most general formalisms, thoroughly defined and for which several interesting compositionality results are already available.

The state space of a TIOA is defined as a set of possible valuations of a set of variables with arbitrary types. The state evolves either by discrete transitions or by *trajectories*. A discrete transition instantly changes the state (i.e. variable valuations) and is labelled with an action that may be internal, an *input* or an *outputs*. Trajectories change the state continuously during a time interval. The behaviour of a TIOA is described by an *execution fragment* which is a finite or infinite sequence alternating trajectories and discrete transitions. The *visible* behaviour of a TIOA is described by a *trace*, which is a projection of an execution trace onto visible actions (inputs and outputs) and in which, from trajectories, only the information about the elapsed time is kept, and information about the variable valuations is abstracted away. For full definitions of all these notions, the reader is referred to [5].

There are two main differences between the TIOA of [5] and our variant:

- [5] allows general functions to be used as trajectories. We restrict ourselves to the identity function for clocks, and to the constant functions for discrete variables. This restriction makes the model expressiveness equivalent to that of Alur-Dill timed automata [1], and will be important later on as it opens the possibility to automatically verify simulation relations between automata (simulation is undecidable for the TIOA of [5]). It also simplifies the presentation of examples, since trajectories are then fully determined by their domain, so we simply use the interval J to represent the trajectory. However, this hypothesis is not needed for proving the compositionality results in section 4.
- In addition to *inputs* and *outputs*, we allow for another type of *visible* actions; this is because, in [5], when composing two automata, an *output* of one matched by an *input* of the other become an *output* of the composite, which does not correspond to our needs when using the TIOA for defining the semantics of usual modelling languages like SysML. As we will show, in order for contract dominance to work, we still need the resulting action to be *visible* in traces, hence the necessity for an additional type of actions.

In the following, for a TIOA A, we denote I_A its set of inputs, O_A its outputs, V_A its visible actions, H_A its internal actions, $E_A = I_A \cup O_A \cup V_A$ and

$A_A = E_A \cup H_A$. The parallel composition operator for TIOA, defined similarly to [5], is denoted by $\|$. We sometimes use the term *component* instead of TIOA, interchangeably.

As in [5], we will use trace inclusion as the refinement relation between components:

Definition 1 (Comparable components). *Two components K_1 and K_2 are comparable if they have the same external interface, i.e. $E_{K_1} = E_{K_2}$.*

Definition 2 (Conformance). *Let K_1 and K_2 be two* comparable *components. K_1 conforms to K_2, denoted $K_1 \preceq K_2$, if $traces_{K_1} \subseteq traces_{K_2}$.*

The conformance relation is used in the definition of refinement under context and for verifying the satisfaction of the system's properties by the top contract: $A \parallel G \preceq \varphi$, where $A \parallel G$ and φ have the same interface. It can be easily shown that conformance is a preorder. The following useful compositionality result, presented in the Timed Input/Output Automata theory of [5], can be easily extended to our variant of TIOA:

Theorem 1 (Composability theorem 8.5 of [5]). *Let I and S be two comparable components with $I \preceq S$ and E a component compatible with both I and S. Then $I \parallel E \preceq S \parallel E$.*

4 Contracts for Timed Input/Output Automata

In this section we provide the definitions for TIOA contracts and the relations described in Section 2, and we list the properties that have been proved upon these and that make contract-based reasoning possible.

Definition 3 (Environment). *Let K be a component. An environment E for the component K is a timed input/output automaton for which the following hold: $I_E \subseteq O_K$ and $O_E \subseteq I_K$.*

Definition 4 (Contract). *A contract for a component K is a pair (A, G) of TIOA such that $I_A = O_G$ and $I_G = O_A$ (i.e. the composition pair $A \parallel G$ defines a closed system) and $I_G \subseteq I_K$ and $O_G \subseteq O_K$ (i.e. the interface of G is a subset of that of K). A is called the* assumption *over the environment of the component and G is called the* guarantee.

Definition 5 (Closed/open component). *A component K is closed if $I_K = O_K = \emptyset$. A component is open if it is not closed.*

In the following, closed components result from the composition open components with complementary interfaces.

The *refinement under context* relation verifies that, given an environment compatible with two components, one component is a refinement of the other in the specified environment. We define this relation with respect to conformance. Since we want to take into account interface refinement between the components

and conformance imposes comparability, we have to compose each member of the conformance relation with an additional timed input/output automaton such that they both define closed comparable systems.

Definition 6 (Refinement under context). *Let K_1 and K_2 be two components such that $I_{K_2} \subseteq I_{K_1} \cup V_{K_1}$, $O_{K_2} \subseteq O_{K_1} \cup V_{K_1}$ and $V_{K_2} \subseteq V_{K_1}$. Let E be an environment for K_1 compatible with both K_1 and K_2. We say that K_1 refines K_2 in the context of E, denoted $K_1 \sqsubseteq_E K_2$, if*

$$K_1 \parallel E \parallel E' \preceq K_2 \parallel E \parallel K' \parallel E'$$

where

- *E' is a TIOA defined such that the composition $K_1 \parallel E \parallel E'$ is closed. E' consumes all outputs of K_1 not matched by E and may emit all inputs of K_1 not appearing as outputs of E.*
- *K' is a TIOA defined similarly to E' such that the composition $K_2 \parallel E \parallel K' \parallel E'$ is closed and comparable to $K_1 \parallel E \parallel E'$.*

The complete formal definition of E' and K' appears in the extended version of this paper.

The particular relationship required between the interfaces of K_1 and K_2 in the above definition is due to the fact that both K_1 and K_2 can be components obtained from composition, e.g. $K_1 = K_1' \parallel K_3$ and $K_2 = K_2' \parallel K_3$, where $I_{K_2'} \subseteq I_{K_1'}$, $O_{K_2'} \subseteq O_{K_1'}$ and $V_{K_2'} \subseteq V_{K_1'}$ (this happens in particular when K_2' is a contract guarantee for K_1'). Then, by composition, actions of K_3 may be matched by action of K_1' but have no input/output correspondent in K_2'. This case also imposes the term $V_{K_1} \cap O_{K_2}$ for the inputs of K', since the additional outputs of K_2 may belong to a different component, and the term $V_{K_1} \cap I_{K_2}$ for the outputs of K'.

Theorem 2. *Given a set $\mathcal{K}$ of* comparable *components, and given a fixed context E for that interface, refinement under context $\sqsubseteq_E$ is a preorder over $\mathcal{K}$.*

The following, required to allow reasoning with contracts, as shown in [12], hold in our framework:

Theorem 3 (Compositionality of refinement under context). *Let K_1 and K_2 be two components and E an environment compatible with both K_1 and K_2 such that $E = E_1 \parallel E_2$. If $K_1 \sqsubseteq_{E_1 \parallel E_2} K_2$ then $K_1 \parallel E_1 \sqsubseteq_{E_2} K_2 \parallel E_1$.*

Theorem 4 (Soundness of circular reasoning). *Let K be a component, E its environment and $C = (A, G)$ the contract for K such that K and G are compatible with each of E and A. If $traces_G$ is* closed under limits *and* closed under time-extension*, $K \sqsubseteq_A G$ and $E \sqsubseteq_G A$ then $K \sqsubseteq_E G$.*

The definitions of closure under limits and closure under time extension for a set of traces are those given in [5]. Closure under time extension informally means that any trace can be extended with time passage to infinity. By making these hypotheses on G, G can only express safety properties on K and cannot impose stronger constraints on time passage than K.

We define *contract satisfaction* based on refinement under context:

Definition 7 (Contract satisfaction). *A component K satisfies (implements) a contract $C = (A, G)$, denoted $K \models C$, if and only if $K \sqsubseteq_A G$.*

Definition 8 (Contract dominance). *Let C be a contract with the interface $\mathcal{P}$ and $\{C_i\}_{i=1}^n$ a set of contracts with the interface $\{\mathcal{P}_i\}_{i=1}^n$ and $\mathcal{P} \subseteq \bigcup_{i=1}^n \mathcal{P}_i$. Then $\{C_i\}_{i=1}^n$ dominates C if and only if for any set of components $\{K_i\}_{i=1}^n$ such that $\forall i,\ K_i \models C_i$, we have that $(K_1 \parallel K_2 \parallel \cdots \parallel K_n) \models C$*

Based on theorems 2, 3 and 4, the following theorem, which is a variant of Theorem 2.3.5 from [12] holds:

Theorem 5 (Sufficient condition for dominance). *$\{C_i\}_{i=1}^n$ dominates C if $traces_A$, $traces_G$, $traces_{A_i}$ and $traces_{G_i}$ are closed under limits and under time-extension and*

$$\begin{cases} G_1 \parallel ... \parallel G_n \sqsubseteq_A G \\ \forall 0 \leq i \leq n. \quad A \parallel (G_1 \parallel ... \parallel G_{i-1} \parallel G_{i+1} \parallel ... \parallel G_n) \sqsubseteq_{G_i} A_i, \end{cases}$$

The above theorem specifies the proof obligations that need to be discharged in order to be able to infer dominance in Step 2 of the verification methodology described in Section 2.

5 A toy example

Figure 2 presents a small example of system composed of three communicating components. The notation is not fully detailed here but should be relatively straightforward. The complete version of the paper provides all the missing detail. We are interested in automata communicating by asynchronous messages. An automaton is contained within a frame, arrows between frames represent messages that are *output* by one automaton and *input* by the other. It is assumed that each automaton has an implicit variable *queue* which stores the incoming messages. The input-enabledness property of TIOA is well suited here: in any state, the automaton can *input* a message and store it in the queue; these *input* transitions are not represented in the figure.

Outputs of a message m are denoted $!m$, consumption of a message m when at the head of the queue is denoted $\downarrow m$. i, j and integer variables while x, y are clocks. We use *urgency* labels to implicitly constrains the set of trajectories starting in a state, like in TA with urgency [2] : *eager* transitions with no guard restrict the set of trajectories to point trajectories only, *eager* transitions with a clock guard restrict the set of trajectories so that they end in the point where the guard becomes true, while *lazy* transitions do not add any restrictions (time may elapse indefinitely).

The system K contains three components. K_1 sends a message a to the environment and a message p to K_2, then awaits for a q signal from K_2. If q is received before a deadline δ_1, K_1 emits a again, otherwise it goes back to the initial state when q is received. In addition, K_1 counts the number of a's emitted (in i), and can answer a message m with a message $n(i)$.

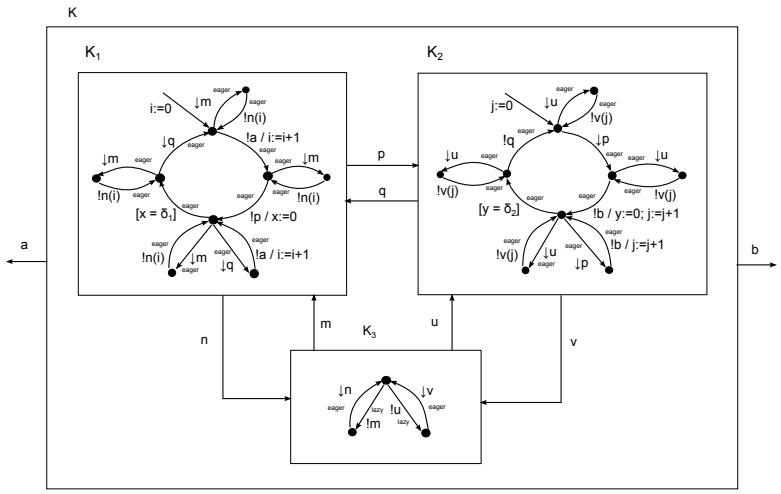


Fig. 2: A system of three communicating components.

K_2 waits for p then sends a signal b to the environment. After that, it waits for δ_2 time units and sends q to K_1; if p is received during this time, b is emitted again. K_2 counts the number of b's emitted (in j), and can answer a message u with a message $v(j)$.

K_3 sporadically sends m and u to K_1 respectively K_2.

The interesting property of this system is that, if $\delta_1 < \delta_2$, then the composition emits a sequence alternating a's and b's, represented in Figure 3. K_3 does not play any role in this property, but hinders its verification if one tries to use model-checking directly on $K_1 \parallel K_2 \parallel K_3$ as exchanges of m, n, u, v will largely contribute to the global combinatorial explosion.

Figure 4 presents a the contracts for the two components K_1 and K_2 which can be used for proving the property.In the case of K_1, the assumption over the environment sends q after at least δ_1 time units since p is received and the component guarantees that consecutive a are separated by an input of q. For K_2, the environment guarantees that it will send p only after receiving a $q$and the component guarantees a delay of δ_2 time units between an output of b and an output of q. For K_3 the contract is given by two empty automata since we don't need any assumption/guarantee on K_3 for proving φ.

The first step of the verification, as presented in Section 2, is to prove that the modelled components satisfy the given contracts. Then, in the second step we prove that the contracts $\{C_1, C_2, C_3\}$ dominate a top-level contract for the system, C, shown in Figure 5. This contract guarantees that a's and b's alternate

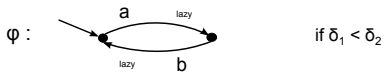


Fig. 3: The property that our example has to satisfy.

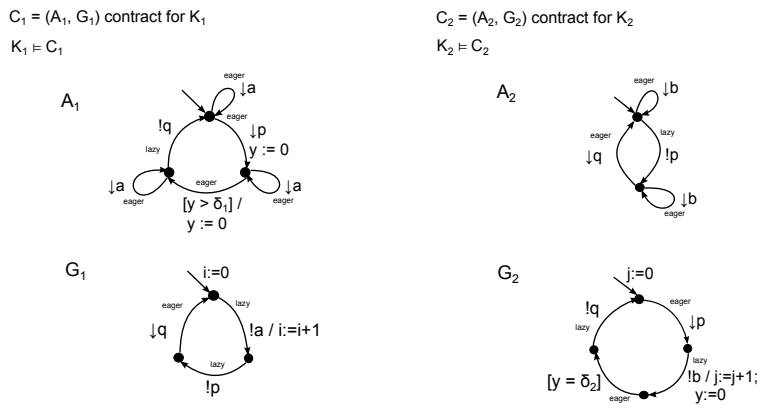


Fig. 4: Contracts for the components K_1 and K_2 of the running example.

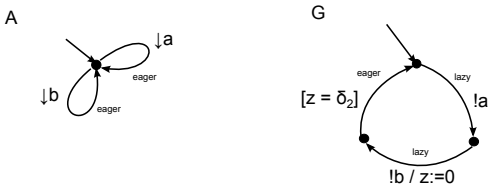


Fig. 5: Top-level contract for K.

with at least a δ_2 delay between them. No assumption is made on the environment. Verifying dominance consists in verifying several refinement under context relations: (1) $G_1 \parallel G_2 \sqsubseteq_A G$, (2) $A \parallel G_1 \sqsubseteq_{G_2} A_2$ and (3) $A \parallel G_2 \sqsubseteq_{G_1} A_1$ (note that we dropped G_3 and A_3 which are empty). The compositions involved have in principle a much smaller state space than the original system K. The last step in the verification of a system model is to prove that the top contract satisfies the global property, i.e. $A \parallel G \preceq \varphi$. All these proof obligations are relatively easy to check manually.

6 Conclusions and future work

We have presented a contract framework for Timed Input/Output Automata and results which allow contract-based reasoning for verifying timed safety properties of systems of TIOA components. For the moment, the method is demonstrated on a small toy example, and many steps of the method remain manual. For example, for the sake of generality, the conformance relation used in the definition of contract satisfaction and in the proof obligations for dominance is TIOA trace inclusion. However, for practical systems one can verify the existence of a simulation, which implies trace inclusion, and for which an efficient automated procedure exists [14].

In addition to other reasons for using contracts mentioned in the introduction, we believe that contract-based reasoning can potentially alleviate the problem

of combinatorial explosion for the verification of large systems. Actually our motivation for exploring contracts is driven by potential applications in system engineering using SysML [9]. In [3] we have presented a case study of an industrial-scale system and we have sketched a proof method for a core property of that system which would use contracts, but which remained to be done. The present work is a first step towards introducing contracts in SysML and providing a full solution to that problem. However, a lot of work still remains: define a suitable syntax for contracts in SysML, define the semantic mapping between the SysML components and contracts and their TIOA counterparts, provide the automatic verification support for contract satisfaction and dominance based on simulation checking, and finally provide quantitative evidence for the efficiency of contract-based versus direct verification.

References

1. Rajeev Alur and David L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, 1994.
2. Sébastien Bornot and Joseph Sifakis. An algebraic framework for urgency. *Information and Computation*, 163, 2000.
3. Iulia Dragomir, Iulian Ober, and David Lesens. A case study in formal system engineering with SysML. In *Engineering of Complex Computer Systems (ICECCS), 2012 17th IEEE International Conference on*, july 2012.
4. E. Allen Emerson and Edmund M. Clarke. Characterizing correctness properties of parallel programs using fixpoints. In *ICALP*, volume 85 of *LNCS*. Springer, 1980.
5. Dilsun K. Kaynar, Nancy Lynch, Roberto Segala, and Frits Vaandrager. *The Theory of Timed I/O Automata - Second Edition*. Morgan & Claypool Publishers, 2010.
6. K. Larsen, U. Nyman, and A. Wasowski. Interface input/output automata. In *FM 2006: Formal Methods*, volume 4085 of *LNCS*. Springer, 2006.
7. RTCA Inc. Software Considerations in Airborne Systems and Equipment Certification. Document RTCA/DO-178C, 2011.
8. Bertrand Meyer. Applying Design by Contract. *Computer*, 25(10):40–51, October 1992.
9. OMG. Object Management Group – Systems Modeling Language (SysML), v1.1. Available at http://www.omg.org/spec/SysML/1.1/, 2008.
10. D.L. Parnas and D.M. Weiss. Active design reviews: Principles and practices. In *Proceedings, 8th International Conference on Software Engineering (ICSE), London, UK, August 28-30, 1985*. IEEE Computer Society, 1985.
11. Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in cesar. In *Symposium on Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351. Springer, 1982.
12. Sophie Quinton. *Design, vérification et implémentation de systèmes à composants*. PhD thesis, Université de Grenoble, 2011.
13. Sophie Quinton, Susanne Graf, and Roberto Passerone. Contract-Based Reasoning for Component Systems with Complex Interactions. Technical Report TR-2010-12, VERIMAG, 2010. http://www-verimag.imag.fr/TR/TR-2010-12.pdf.
14. Farn Wang. Symbolic simulation-checking of dense-time automata. In *FORMATS*, volume 4763 of *LNCS*, pages 352–368. Springer, 2007.

Session du groupe de travail COSMAL

Composants Objets Services : Modèles, Architectures et Langages

A Dynamic Component Model for Cyber Physical Systems

Francois Fouquet, Olivier Barais,
Noel Plouzeau, Jean-Marc Jezequel
IRISA, University of Rennes1, France
firstname.name@irisa.fr

Brice Morin Franck Fleurey
SINTEF ICT, Oslo, Norway
firstname.name@sintef.no

ABSTRACT

Cyber Physical Systems (CPS) offer new ways for people to interact with computing systems: every thing now integrates computing power that can be leveraged to provide safety, assistance, guidance or simply comfort to users. CPS are long living and pervasive systems that intensively rely on microcontrollers and low power CPUs, integrated into buildings (e.g. automation to improve comfort and energy optimization) or cars (e.g. advanced safety features involving car-to-car communication to avoid collisions). CPS operate in volatile environments where nodes should cooperate in opportunistic ways and dynamically adapt to their context. This paper presents μ-Kevoree, the projection of Kevoree (a component model based on models@runtime) to microcontrollers. μ-Kevoree pushes dynamicity and elasticity concerns directly into resource-constrained devices. Its evaluation regarding key criteria in the embedded domain (memory usage, reliability and performance) shows that, despite a contained overhead, μ-Kevoree provides the advantages of a dynamically reconfigurable component-based model (safe, fine-grained, and efficient reconfiguration) compared to traditional techniques for dynamic firmware upgrades.

Categories and Subject Descriptors

D.2 [**Software Engineering**]; D.2.8 [**Software Engineering**]: Software Architectures—*Domain-specific architectures; Languages*

Keywords

Component-based software engineering ; Autonomic computing ; Embedded software ; Software architecture

.

1. INTRODUCTION

Over the last decades, the Internet has undergone dramatic changes, moving from a rather static Internet of Content, to an always more complex, dynamic and ubiquitous mix of Internet of People, Services (IoS), and Things (IoT) [1]. Based on this infrastructure, which ranges from large datacenters and cloud servers to an heterogeneous and ever growing set of things (smartphones, sensors, etc) operated by resource-constrained CPUs and microcontrollers, Cyber Physical Systems (CPS) have emerged. CPS are long living and pervasive systems that rely intensively on microcontrollers and low power CPUs, integrated into buildings and cities (automation to improve comfort, safety and energy optimization), cars (advanced safety features involving car-to-car communication to avoid collisions), and so on.

CPS operate in volatile environments where nodes should cooperate in opportunistic ways and dynamically adapt to their context. In a car-2-car scenario, 2 cars (or more) approaching the same intersection should be able to synchronize in a reasonably short delay to share information about their own context and configuration, then take distributed decisions *e.g.* on the precedence order to cross the intersection. In a building automation scenario, users working or living in the building should be able to customize their working or living environment according to their desires and needs, *e.g.* using their smartphones or tablets to adjust the intensity of the lights according to the ambient light, etc. In a factory chain scenario, robots operating on the chain should be able to adapt and cope with failures, instead of shutting down all the chain when a failure is detected. Depending on the context, it is necessary to dynamically adapt both the software and the way the CPS are configured, as things are containers that can host services.

Dynamic adaptation, pursuing IBM's vision of autonomic computing, is a very active area since the late 1990's - early 2000's [20]. However, many existing techniques concentrate on the adaptation of rather powerful nodes, which are typically able to run a Java Virtual Machine. Adaptation of resource-constrained devices such as microcontrollers has received less attention. These resource constraints prevent the use of standard operating systems, middlewares and frameworks, making the design of adaptive software for microcontroller a challenging task. In practice, microcontroller code is most of the time developed by using low-level programming languages and by following ad-hoc manual trial and error processes; these processes includes extensive testing of the resulting software in its target environment. While this might be acceptable to build static, dedicated applications,

this is not a practical solution for CPS, because CPS operate in an open and dynamic environment, where the target environment cannot be foreseen at design-time.

Kevoree leverages and extends state-of-the-art approaches to scale CBSE principles horizontally (distribution between a large set of nodes) and vertically (from cloud-based nodes to microcontrollers). This paper focuses on μ-Kevoree, a mapping of Kevoree concepts for microcontrollers. μ-Kevoree pushes dynamicity and elasticity concerns directly into resource-constrained devices. In particular, this paper details the challenges of mapping such a large component model onto microcontroller-based architectures. We explain the tradeoffs that were used to obtain a useful solution coping with stringent resource constraints. This dynamic component model for resource-constrained systems has been thoroughly benchmarked against key criteria that are specific to the embedded software domain (memory usage, reliability and performance). Our model has also been applied to a real-life case study. The evaluation of μ-Kevoree for these key criteria show that, despite a contained overhead, μ-Kevoree provides a dynamically reconfigurable component-based model (safe, fine-grained, and efficient reconfiguration) with a limited overhead with respect to static approaches.

This paper is organized as follows. Section 2 presents the Kevoree component model and Section 3 details the challenges of mapping dynamic component model concepts to resource-constrained microcontrollers. Section 4 then explains how we ported Kevoree to these challenging platforms, and details the necessary tradeoffs. This new version of Kevoree is validated in Section 5 through a set of atomic benchmarks. Section 6 discusses the result and presents related work and Section 7 concludes and draw some perspectives to be addressed in future work.

2. BACKGROUND

2.1 Kevoree at a glance

Kevoree [1] is an open-source dynamic component model, which relies on models at runtime [6] to properly support the dynamic adaptation of distributed systems. Models@runtime basically pushes the idea of reflection [23] one step further by considering the reflection layer as a real model that can be uncoupled from the running architecture (e.g. for reasoning, validation, and simulation purposes) and later automatically resynchronized with its running instance.

Kevoree has been influenced by previous work that we carried out in the DiVA project [23]. With Kevoree we push our vision of models@runtime [22] farther. In particular, Kevoree provides a proper support for distributed models@runtime. To this aim we introduce the *Node* concept to model the infrastructure topology and the *Group* concept to model semantics of inter node communication during synchronization of the reflection model among nodes. Kevoree includes a *Channel* concept to allow for multiple communication semantics between remote *Components* deployed on heterogeneous nodes. All Kevoree concepts (Component, Channel, Node, Group) obey the object type design pattern [18] to separate deployment artifacts from running artifacts. Kevoree supports multiple kinds of execution node technology (e.g. Java, Android, MiniCloud, FreeBSD, Arduino, ... [1]).

[1] http://www.kevoree.org

2.2 Dynamic Adaptation with Kevoree

Kevoree aims at providing advanced adaptation capabilities to different types of nodes:

- **Level 1: Parametric adaptation.** Dynamic update of parameter values, e.g. change of sampling rate in a component that wraps a physical sensor (adaptation of instance properties).
- **Level 2: Architectural adaptation.** Dynamic addition or removal of bindings or components, e.g. replication of software components and channels on different nodes to perform load balancing (adaptation of instances graph).
- **Level 3: Dynamic provisioning of types.** Hot deployment of component types that were not foreseen before the initial deployment of the system. This allows for system evolution by enabling parametric and architectural reconfigurations, including management of instances for types that are added and managed dynamically (adaptation of types).
- **Level 4: Adaptation for remote management.** Nodes supporting level 4 adaptation participate in a remote management layer, which supervises less powerful nodes. This layer monitors remote nodes by requesting their current Kevoree model; the layer triggers dynamic adaptation of nodes by sending precomputed reconfiguration scripts to them. This remote adaptation process supports seamless management of less powerful nodes by a more powerful one, which has enough resources to build and evaluate new and appropriate configurations.

The adaptation engine relies on a model comparison between two Kevoree models to compute a script for a safe system reconfiguration; execution of this script brings the system from its current configuration to the new selected configuration [23]. Model comparison yields a delta-model defining changes (using CRUD operations) that should be applied on the source model to obtain the target model. Planification algorithms [4] use this delta-model as input in order to defined an efficient schedule of the adaptation steps. The delta-model is finally compiled into a Kevoree script. The Kevoree Script language (KevScript for short) is a core language for describing reconfiguration. KevScript is comparable to FScript for Fractal Component Model [11]. Execution of a KevScript directly adapts a Kevoree system, without the need for a full Kevoree model definition. Such adaptation scripts are written by designers, or they can be generated by automated processes (*e.g.* within a control loop managing the Kevoree system).

3. MAPPING KEVOREE ADAPTATION CONCEPTS ON MICROCONTROLLERS

3.1 Challenges

Dynamic adaptation is a key concept to build advanced CPS able to adapt to their context and to user needs. Models at runtime is an efficient approach to manage the complexity of dynamic adaptation [23] by providing control and abstraction over reflection mechanisms. Applying reflection techniques is rather straightforward on fully grown component or service models such as OSGi, or directly on top of modern object-oriented languages such as Java, as long as

the execution hardware is powerful enough to run a virtual machine. The embedded software sensing and acting on the physical world (via hardware components) should be able to adapt to the needs of different (and potentially concurrent) services running on powerful nodes (in the cloud, on tablets, etc). Some services will for example subscribe to temperature alerts (the sensors being responsible to notify the service when a threshold has been reached), and then reconfigure the sensors to send almost continuous data, so that the service can precisely monitor the evolution of the temperature.

Applying models at runtime (or any dynamic adaptation technique) on execution nodes with scarce resources (e.g. microcontroller-based computation nodes) is much more difficult, for the following reasons:

1. **Downtime**: Microcontrollers often host the software that controls physical devices directly. Rebooting or freezing these microcontrollers may have severe consequences if the microcontrollers control safety critical devices, or unpleasant and noticeable effects if they control comfort devices.
2. **Volatile memory usage** (RAM): Dynamic memory allocation is the cornerstone that enables dynamic adaptation. Microcontrollers usually embed only of few kB of RAM, and this size limitations prohibits storing multiple configurations in memory at the same time.
3. **Persistent memory usage**: Persistent memory is required to ensure that the adaptation process has transaction-like properties, allowing recovery of microcontroller's state in case of a reboot after failure. EEPROM is a common type of persistent memory embedded into microcontrollers, usually with a very limited size. This type of memory also has a limited lifetime in term of numbers of writing operations. Similar to Solid State Disk [2], writes to EEPROM should be distributed among memory cells to optimize the lifetime of the overall memory.
4. **Recovery**: The ability to recover is critical for embedded systems, which are subject to failures (e.g. a temporary loss of power). Microcontrollers should reboot and restore their last configuration quickly enough to keep pace with configuration evolutions of the overall architecture.

CPS relying on a large set of autonomous sensors have cost and energy constraints that calls for cheap and power-efficient platforms able to run for long period of times with minimum on-site maintenance (*e.g.*, battery replacement). This is particularly true in the environmental monitoring domain: off-shore oil spills monitoring, flood prediction [22], air quality monitoring or radiation monitoring, for the following reasons[2]:

1. Their simplified architecture is robust and predictable: microcontrollers can operate by a wide range of temperature (typically -40 to 85 degrees Celcius), humidity, power supply, and have fixed number of cycles to execute a given operation.
2. Their energy needs (and generated heat) are very low: An 8-bit microcontroller running at 32kHz typically consumes less than 0,05W (less than 0,5W at 1MHz) excluding the need for any radiator. They can thus run for very long time on battery.
3. Their simplified architecture allows for mass production, making microcontrollers very cheap to deploy even in large numbers.

Compared to full-fledged computation nodes, cheap microcontrollers suffer from an adaptation overhead that stems from their hardware technology in terms of adaptation time or memory wear: dynamic provisioning of component types requires writing a program in flash memory. Therefore, implementing Kevoree concepts for microcontrollers nodes relies on a precise trade-off between flexibility and typical exploitation costs. Finding a lightweight solution for each of reconfiguration level described above is one of the main challenges of μ-kevoree.

3.2 Case study

We will use a smart building case study to validate Kevoree on a set of heterogeneous nodes, including of course some microcontrollers. Different systems (relying on proprietary devices and protocols) are usually deployed in buildings to manage different aspects of the building automation, in particular comfort (lighting, air conditioning, etc), safety and security (smoke and fire detection, sprinklers, etc). The degree of flexibility offered by a building automation system is often very poor.

- These systems rely on fixed topology of communication channels. Sensors and actuators often need to be physically coupled, hindering any future reconfiguration or evolution of the system. For example, a motion sensor will trigger all the lights of the corridor.

- The architecture is organized around a central server. When the devices are not physically coupled, they usually communicate through a central server, to execute the event-driven rules that orchestrate the system. Even though updating these rules is possible, to adapt the behavior of the system, this requires access to the central server.

The goal of Kevoree is to seamlessly distribute both the business logic and the dynamic adaptation capabilities on heterogeneous nodes ranging from powerful servers, to tablets, and to simple devices operated by microcontrollers. On a day-to-day basis, this would allow users to configure and reconfigure their offices on-the-fly from a smartphone, (e.g. to define a lighting environment according to the ambient luminosity, temperature, etc), while some other concerns would be managed by a central server (e.g. to turn the cameras on at night). In a crisis situation, this kind of seamless distribution would allow emergency services to cope with the failure of some nodes. Firemen could still access the data provided by low-level sensors, and compute meaningful context information on a tactical decision system despite the loss of a nodes.

4. DYNAMIC ADAPTATION FOR MICROCONTROLLERS

This section describes how Kevoree concepts are mapped to Arduino nodes. Arduino[3] is an open-source hardware and

[2]See for example http://www.atmel.com/Images/doc2545.pdf for detailed facts about microcontrollers

[3]http://www.arduino.cc

software electronics prototyping platform based on an 8-bits AVR microcontroller. Arduino boards can be connected to a set of sensors and actuators and programmed in languages from the C/C++ family. While we have chosen Arduino to implement our μ-Kevoree approach, it can be applied easily to other microcontroller families (PIC, ARM, etc).

Firmware implementations are often coded manually in C using a trial and error process, with an intensive manual and automated test-based validation. More advanced techniques (such as the MDE techniques proposed by ThingML[4] [15]) aim at generating static source code for microcontrollers. Such techniques can easily be leveraged to generate the internal code of component, which is not the scope of Kevoree. Microcontroller firmware puts a strong emphasis on resource usage (such as memory, CPU and energy needs) and reliability: microcontrollers can run for long periods of time and recover in case of power or connectivity loss. We acknowledge that these properties are critical, and the benefits provided by dynamic adaptation capabilities should not jeopardize them. Our work aims at breaking the static nature of code generation while preserving all benefits of low level code design.

To this aim our approach clearly separates structure and behavior: component type behaviors are currently implemented manually in C or in the Wiring language, using state of the art practice. One core contribution of our approach is the definition of a proper abstraction system to automate management of the adaptation logic of microcontrollers, by making their business logic a separate concept. Moreover, having a clear component structure with well defined inputs and outputs also eases testing at a more abstract level.

4.1 μ-Kevoree

This sub-section describes how the main concepts of Kevoree have been ported onto microcontrollers. μ-Kevoree is totally aligned and compatible with the exiting Java and Android versions.

Types. In Kevoree component types and channel types encapsulate business logic; they are generated as C structures. *Provided* ports of component types are mapped to methods, so that client components (which require ports of the same type) can invoke these methods and eventually push data. Kevoree properties that can be dynamically updated are simply mapped onto local variables contained by this structure. A local scheduler prevents concurrent calls on these variables. *Required* ports are generated as local structures, which can optionally refer to a bound channel instance. Similarly, channel types are generated as plain C structure. Outgoing channel bindings are generated as an internal array structure, enabling dynamic allocation and storage of external provided ports references.

Asynchronous message passing. As in Kevoree's implementations for Java and Android nodes, μ-Kevoree maps each port and channel onto an actor. More precisely, a FIFO queue is generated in front of each protected method. A dispatcher (local to each component) is then in charge of dispatching messages pushed on these queues to the correct method. This local scheduler is driven by a global scheduler described below.

Instance scheduler On each node a global instance scheduler is responsible for keeping its node in a consistent state by applying the following balance strategy:

- Periodic execution: the global scheduler periodically invokes the local scheduler of each component instance that has declared a periodic execution;
- Triggered execution: the global scheduler invokes the local scheduler of each component that has a non empty message queue.

The global scheduler also periodically checks for external messages related to dynamic adaptation, as described in the next two sub-sections.

4.2 Firmware flash to handle major evolutions

Flashing a microcontroller's firmware and then rebooting the device is an easy way to implement adaptation of a microcontroller node, by replacing the implementation entirely. This adaptation technique is acceptable in some specific and controlled contexts (initial production, on-site maintenance, etc), since the device is physically connected to a more powerful node using a communication link with broad bandwidth (e.g. wired link). In this case, flashing a controller's memory is rather safe (provided that the code of the firmware is safe) and also reasonably fast: flashing the entire memory by uploading the new firmware and then rebooting the device takes a few seconds only. However, this technique is problematic when the devices are deployed remotely. Flashing the firmware over-the-air is a hazardous manipulation: firmwares are typically bulk data (compared to other data usually transmitted on wireless links) and communication errors are more likely to occur; this requires advanced protocols to cope with error handling. In practice, this approach impacts significantly the time needed to install a new firmware.

Our approach limits flashing the full firmware to cases where new component types need to be deployed. In this regard C-based microcontrollers do not provide the same flexibility than Java/OSGi nodes with respect to dynamic provisioning and class loading. This is typically required for the initial deployment of the system where all the planned component types are provisioned, or for major evolutions of the system (e.g. to handle a new type of device not foreseen before the initial deployment). In all other cases such as reconfigurations of component instances for instance, our approach performs a partial flash memory update.

4.3 Seamless dynamic adaptation of microcontrollers

Following the principles of *model@runtime*, our dynamic adaptation process is fully automated, saving designers from writing low-level adaptation scripts or from entangling adaptation logic with business logic. Prior to any adaptation, all necessary checks on the new configuration are performed on the target model. Since microcontroller nodes have limited computational power, configuration checks are performed on more powerful, Java or Android based nodes. These checks aim at detecting a mismatch between the planned configuration and the physical hardware possibilities. After this validation step, the configuration is used as input for a generator algorithm, which computes a reconfiguration script. This script is then transmitted in a compact form to dependent microcontrollers. As communication errors are frequent in wireless sensors networks, we avoid problematic microcontroller states inconsistencies by implementing a roll-back based recovery mechanism.

[4] www.ThingML.net

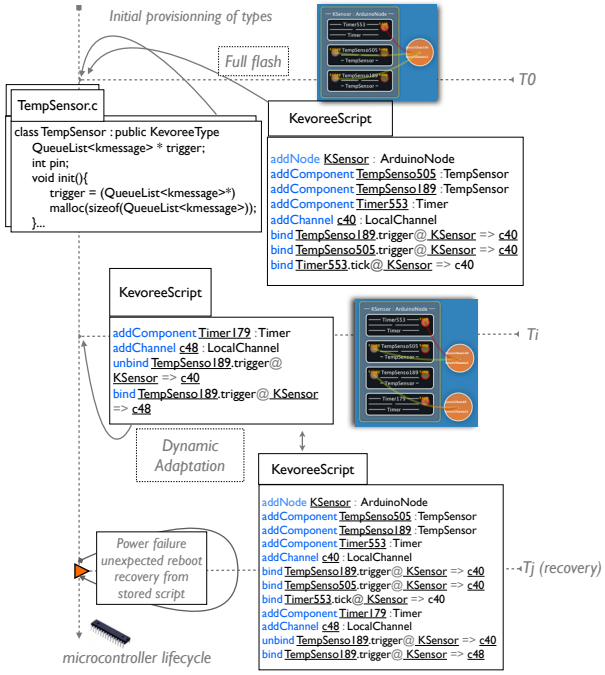


Figure 1: Overview of micro-Kevoree

The following table lists the dynamic adaptation levels supported by the different Kevoree node technologies; the levels are defined at the beginning of Section 2. The most powerful nodes are able to run Java programs and implement all levels of adaptation features. The more constrained are the nodes, the more tradeoffs have to be addressed.

Dynamic adaptation	JavaSE	Android	Arduino
Parametric	+	+	+
Architectural	+	+	+
Dynamic provision.	+	+	+/- firm. flash
Peers management	+	+/- perf. issues	- see Future work

While severely limited resource-wise, Arduino nodes are still able to support almost all dynamic adaptation features. In most cases, complex decisions will be taken by software running on more powerful nodes. Taking adaptation decisions require processing adaptation rules, optimize goals, etc, as microcontrollers usually do not have enough computational power.

Dynamic instantiation framework.

Reflection is a fundamental principle to achieve dynamic adaptation. But the C language has no support of the primitives required to build a full-fledged reflection model; this prevents a straightforward application of the model@runtime technique on microcontroller nodes. We have removed this limitation by generating code to emulate a reflection layer on these microcontrollers. We assume that the number of types is finite when generating the framework. Adding or removing a type then implies regeneration of the whole framework as described in Section 3. With this assumption of a closed type world, we generate a flat reflection layer by using exhaustive pattern matching on instances. We generate methods that take instances as parameter; these methods provide the following basic services:

Algorithm 1 μ-Kevoree core service

```
Function interruptScheduler(),resumeScheduler()
Function setProperty(instanceID,propID,propValue) : Bool
Function addBinding(channelID,componentID,portID) : Bool
Function removeBinding(channelID,componentID,portID) : Bool
Function createInstance(instanceID,typeID) : Bool
Function destroyInstance(instanceID) : Bool
Function exportCurrentState() : KevScript
```

We rely on script to export the state in order to optimize memory consumption. More precisely, every textual representation used to export the reflection model and related to type definitions (*e.g.*, port names) is locally encoded in static flash memory to save dynamic memory.

KevScript embedded interpretation.

An embedded KevScript interpreter uses the flattened reflexivity methods to implement basic services (e.g. instance life cycle, binding and parameters management).

Upon receiving a Kevoree script in a compressed format a microcontroller performs the following tasks:

Algorithm 2 KevScript Interpreter

```
Function interpretScript(script : KevScript)
  interruptScheduler()
  if permanentMemory.size >= PermanentMemoryLimit then
    resetPMemoryIndex()
    writeToPMemory(exportCurrentState())
  end if
  lastRecoveryPoint ← createRecoveryPointInPMemory()
  ∀ st, st ∈ script.statements → writeToPMemory(st)
  if executeFromPMemory(lastRecoveryPoint) then
    closeRecoveryPoint(lastRecoveryPoint)
  else
    rollback(lastRecoveryPoint)
  end if
  resumeScheduler()
```

Scripts are checked independently of the communication context and then stored. The new configuration is committed by an atomic write in the memory once it has been validated, thereby implementing an atomic transaction mechanism. This technique prevents incomplete memory saves. In case of any problems during the KevScript interpretation, a rollback is achieved by a simple reboot of the microcontroller. The Figure 1 gives an overview of this process taking as an example a temperature sensor reconfiguration. At T0 time a full configuration is pushed containing C code and an initial KevScript. At Ti time a KevScript is pushed adding 2 instances. Between Ti and Tj time a power failure occur resulting on a recovery using memory saved KevScript.

5. VALIDATION

This section describes our experimental setups for validating our approach against the criteria identified in Section 3[5]. More precisely, our experiments measure the following parameters:

Downtime: overall time needed by the microcontroller to adapt, including uploading of the new configuration. This metric thus measures the overhead induced by the microcontroller to manage its own state with respect to domain

[5]More details on this experiment can be found `http://blog.kevoree.org/pages/kevoree-for-microcontroller-benchmark`

applications execution time.

Volatile memory usage (RAM): amount of RAM memory dedicated to dynamic allocation of component instances, channels, and bindings. This metrics thus influences the maximum number of component instances, channels and bindings that a microcontroller can manage.

Persistent memory usage: amount of persistent memory used to store reconfiguration scripts and impact of storage strategy on memory life time. Persistent memory types such as the EEPROM embedded in the 8 bits AVR have a limited number of write cycles certified for each byte, thereby limiting the amount of storable data.

Recovery reboot delay: time needed by the microcontroller to reboot and restore its last configuration, after a crash or a loss of power.

We have used realistic configurations to assess our approach and evaluate the overhead induced by our dynamic component-based platform for microcontrollers, compared with static configurations that are updated by flashing the whole memory. All experiments were done using the Kevoree Arduino node implementation [6] running on an Arduino board with an ATMEL AVR 328P microcontroller. This processor embeds 32 KB of flash memory for storing programs, 2 KB of RAM memory and 1 KB of EEPROM. A flash-type memory (microSD) connected via an SPI bus was also used as persistent memory to assess the impact of memory type on results.

The following subsections show our specific experimental protocol and results, while the last subsection will present an industrial use case to validate the seamless integration of μKevoree devices in an existing dynamic architecture.

5.1 Downtime: How long does an adaptation freeze business logic?

Experimental setup. In this experiment we setup five different configurations, similar to the ones presented in the case study (building automation). The corresponding Kevoree models used different numbers of instances to simulate changes between the configuration used at night and a personalized configuration used during the day. More details on these models are available here [6] . In a nutshell, these models were configured with 4 nodes, hosting 0 to 10 instances each. Instances are implemented in C, with 30 lines of code each on average.

In a first step we generated the firmware of our test microcontroller, with code containing all type definitions used in this experiment. This step therefore includes code generation, compilation and writing into flash memory. Moreover, the generated code is automatically instrumented with probes to measure downtime and memory use (EEPROM and SDRAM). This step was repeated delayed of 100 ms with a new configuration that is chosen randomly; each new configuration was dynamically installed to replace the current running configuration. This random reconfiguration step was repeated 500 times. Figure 2 plots the raw data collected in this experiment. The plot on top shows that the RAM usage is constant. The second plot shows the downtime per reconfiguration, and third and bottom plots show downtime and script size respectively.

[6] `http://goo.gl/Xl2z9`

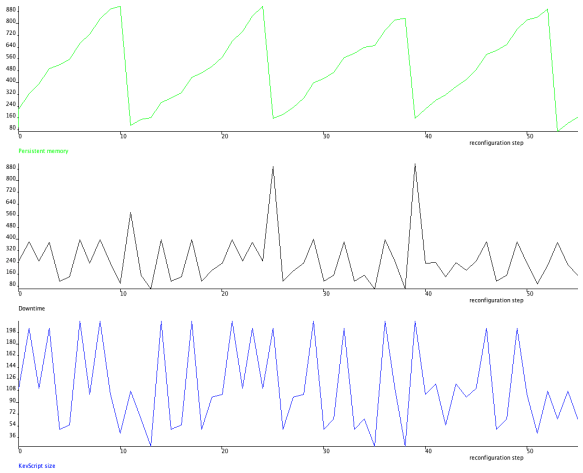


Figure 2: Experiment raw results[7]

Experimental results and analysis. Deploying a configuration by flashing the whole firmware is very costly: the downtime to deploy the initial configuration is 12.208 seconds. This high value comes mainly from the long transfer time of a full firmware but also from the time taken by the default boot loader to perform a full restart of the microcontroller. This value varies in a range of +/-2 seconds.

Results of this first experiment highlight that the size of the reconfiguration script is highly correlated with the downtime time: the Spearman correlation coefficient observed between script size and downtime is higher than 0.9. In addition, the compression algorithm used to decrease the script size in EEPROM has also an impact on downtime. We observed that the execution of this task is directly correlated with higher values of downtimes. This is discussed in the next subsection.

After 500 cycles of reconfiguration we measured the following extrema and mean values:

- minimum downtime of 58 ms, 210 (i.e. 12208 / 58) times faster than static flashing;
- maximum downtime of 916 ms, 14 (i.e. 12208 / 916) times faster than static flashing;
- mean downtime of 235 ms, 52 (12208 / 235) times faster than static flashing.

We used a distribution by percentiles graph for downtime values to better analyze these data, as shown in the following table.

Percentile(%)	0	5	25	50	75	95	100
Downtime (ms)	58	59	139	221	248	398	916

The graph in Figure 3 clearly shows that the downtime values are clustered around 220 ms. 95% of the values are below 400 ms and 75% are below 250 ms. Then, only 5% of the values are above the 400 ms, which is explained by the EEPROM compression step. The lazy compression strategy allows us to limit the number of peaks and keep the maximum value around 200 ms. The highest values for the downtime are systematically linked to a reduction of the EEPROM size (caused by the compression routine). We observed 32 compressions of the EEPROM during the 500 reconfigurations *i.e.*, 6.4% of the reconfigurations trigger a compression so that they can be completely stored into the EEPROM. The maximum value of these 32 downtime peaks is 916 ms, the minimum is 218 ms and the mean value is 580.815 ms. This

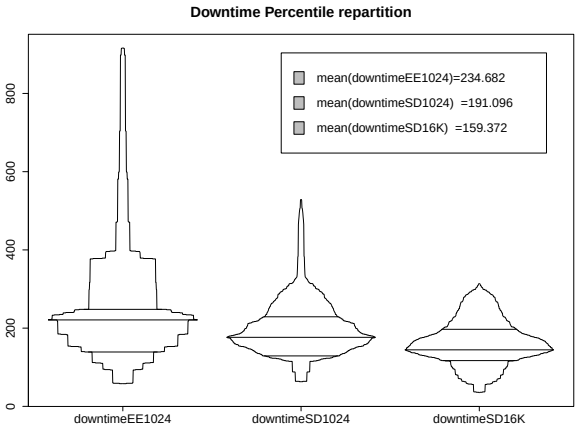


Figure 3: Flash RAM percentile downtime distribution (in ms)

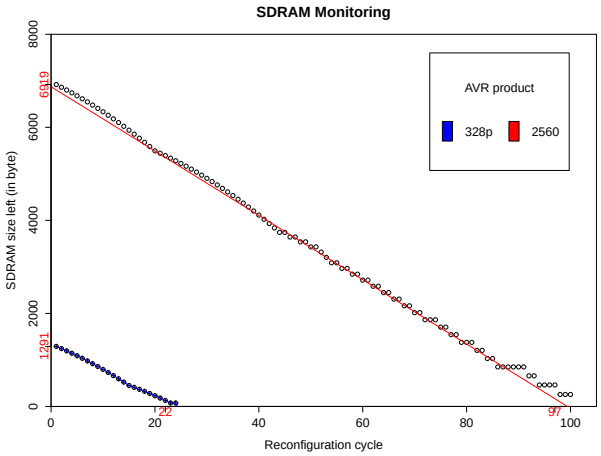


Figure 4: SDRAM capacity experiment

mean value is significantly higher than the mean value of the whole set of 500 reconfigurations (234.682 ms).

Probes also monitored SDRAM during this experiment. Neither memory leaks nor memory fragmentation occurred. Although the SDRAM is stable, it is necessary to check that the overhead induced by the framework actually allows for the dynamic creation of a realistically high number of instances, to match concrete use case needs. Our next experiment aimed at evaluating the capacity (in terms of dynamic instances) of our test microcontroller.

Experimental results and analysis with a different setup. We used the same protocol of 500 iterations, but we replaced the EEPROM with a 2 GB external flash memory (SD card), connected on an SPI bus. This experience is repeated twice: with only 1kb (same size as EEPROM), and with 16kb; results are shown in the following table.

Percentile(%)	0	5	25	50	75	95	100
DowntimeSD 1K(ms)	63	88	129	176	229	324	529
DowntimeSD 16K(ms)	35	56	117	145	197	297	314

Flash memory has a longer initialization time, which explains that the lowest values for the flash experiment are higher than the lowest value in the EEPROM experiment. However, writing speed of flash memory is high, resulting in a homogenization of downtime, which is under 200 ms most of the time. One can notice that the large increase of persistent memory (16 kb) clips the downtime peaks and therefore improves average downtime value. However this does not change the distribution of main values significantly.

5.2 Volatile memory usage: how many instances?

Experimental setup. The purpose of this experiment was to precisely determine the maximum number of instances that can fit into the SDRAM. An initial configuration was created with three instances: a timer, a switch and a default channel. Every 100 ms, the configuration was expanded by adding a new switch instance. Probes were injected to monitor the SDRAM.

Experimental results and analysis.
Figure 4 shows that SDRAM memory is full after 22 cycles, *i.e.* our test microcontroller can manage 25 instances (the 3 initial ones plus 22 additional instances). In practice, the count of devices controlled by one single microcontroller is approximately equal to the number of pins they have. In addition, microcontrollers should be able to run some code to orchestrate these devices and perform some computation. Kevoree is able to manage 25 instances (both for wrapping physical devices and for defining some orchestration) on our test microcontroller (22 pins). Despite a memory overhead, our approach is compatible with current practices.

Experiments results and analysis with a different setup. In this setup we used an ATMEL 2560 (4 KB of SDRAM) as our test microcontroller. The AVR 2560 accepts a load of 100 instances *i.e.* an improvement of+300% instances with +300% SDRAM. Again, the number of instances is larger than the number of pins (80) of the AVR 2560.

5.3 Persistent memory usage: How many certified reconfigurations?

Experimental setup. We used the setup of Section 5.1.

Experimental result and analysis. We observed that $500/32 = 15.625$ reconfigurations can happen before the EEPROM (1 KB) is full, requiring a compaction using a new initial state. As for Solid State Disk [2], write operations to EEPROM should to be distributed throughout the memory in order to distribute wear. Our algorithm writes every byte before computing a new initial state. Each byte of this memory is certified for 100,000 writes [7]. Therefore if we assume 100 reconfigurations every day (which is much more than what is needed in most case studies), each byte of the EEPROM will be written 6.4 times a day on average, ensuring 15,625 days (i.e. about 43 years) of certified lifetime for the EEPROM.

Experiments results and analysis with a different setup.
In average, we can serialize the reconfiguration scripts of our experiment using 16 bytes. Since our algorithm ensures that every byte is written before the memory needs to be compressed, we can use the reasoning of the previous paragraph to compute the lifetime with a larger memory.

[7] http://arduino.cc/en/Reference/EEPROMWrite

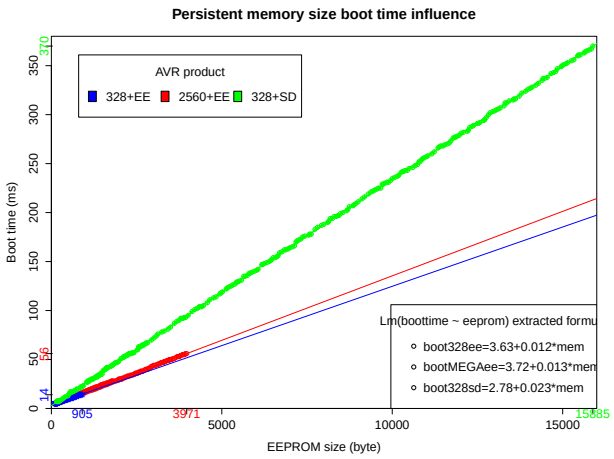


Figure 5: Persistent memory size boot time influence

5.4 Recovery reboot delay: How long to recover?

Experimental setup. This experiment used the configuration set described in Section 5.1, with only 50 cycles of reconfiguration performed every 2 seconds. The microcontroller was physically rebooted between each reconfiguration, and a new probe was inserted to measure the configuration restore time the after booting.

Experimental results and analysis.
Figure 5 displays the results of this second experiment. It appears that in the worst case with this AVR product (when the 1 KB EEPROM memory is almost full) the boot time is approximately 15 ms. In the best case (EEPROM almost empty) the boot time is approximately 3 to 4 ms. This value is mainly due to the slow read speed of the EEPROM embedded in the AVR. However, the time to restore a configuration is reasonable for most use cases, even in the worst case. We can therefore infer that the script size in EEPROM has a small impact on boot time with respect to save time. Therefore the best strategy is to use all available memory.

Experimental results and analysis with a different setup.
We used the same setup, but we replaced the EEPROM with a flash memory (1 KB and 16 KB). Using SD memory instead of EEPROM implies a longer boot time. We noticed a coefficient value of 0.012 for the EEPROM, and of 0.023 for the SD. This comes from the extra computation needed to read and write SD card. Unlike the EEPROM, the communication bus to access the SD flash is external to the microcontroller. The initialization time taken by flash memory configuration is linearly distributed to the limit of 16 KB. Above this size, the initialization time becomes greater than 360 ms. This is significantly higher than the reconfiguration time.

5.5 Discussion

The choice of persistent memory type and size depends on the use case needs. In some cases, there is a real need for traceability, and the history of the system should be kept *e.g.*, for *post mortem* analysis in case of failure. In other cases, performance of adaptation and boot time after a failure is more important. The benchmark developed in this approach can be useful to determine empirically which memory setup to use.

However, using an external memory significantly increases the price of such a platform. In addition, ensuring atomicity of reconfigurations is more difficult because of the asynchrony of transfer protocols. Existing protocols for interaction with SD cards (like MMC) are not suitable for storing reconfiguration scripts. More precisely, these scripts are much shorter (around 25 bytes in our experiments) than the minimum frame size (512 bytes) required by these protocols, and this leads to a significant overhead. In practice, most embedded devices combine EEPROM and flash memories. Kevoree allows designers to combine different memory types for different purposes.

μ-Kevoree overhead. Our framework adds several overhead sources, especially for the management of dynamic instance creation of components and channels. In order to quantify overhead, we measured volatile and program memory sizes on an HelloWorld program, using plain C and a Kevoree firmware setup on a 328P AVR microcontroller. The plain C version left 1842 free bytes after boot sequence, while the Kevoree version left 1604 bytes free. This represents an overhead about 11% of the total available RAM (242 bytes out of 2048). The plain C version used 2.3 KB of firmware memory, while the Kevoree use was 7.3 KB, giving an overhead of about 15% of the total 32kb available. The impact of Kevoree scheduler on processing cycles is highly program dependent, and we are currently experimenting further to compute this overhead.

Our synchronization and communication layer introduced an overhead under 15% on both memories, which is an acceptable value in the case of our IoT application. However, this impact should be evaluated in more depth for hard real-time applications, with a special attention to components needs in terms of processing cycles.

6. RELATED WORK

Software architecture aims at reducing complexity through abstraction and separation of concerns by providing a common understanding of component, connector and configuration [10, 21, 32]. One of the remaining challenges, strengthened by the Future Internet and CPS [24], is to properly manage dynamic architectures. SCA [8] is a standard that highlights modular software architecture concepts. It provides a model for composing applications that follow Service-Oriented Architecture principles. Frascati [29] is a SCA runtime that allows developing highly configurable applications. However, SCA focuses on rather heavy nodes typically able to run a JVM whereas μ-Kevoree also manages the dynamic adaptation of microcontroller-based systems, in addition to Java nodes, with a contained overhead.

Many approaches have highlighted the need for dynamic architectures to implement pervasive computing. A common approach consists in building a middleware to hide the heterogeneity of networks, hardware, operating systems, and programming languages.Rellermeyer *et al.* [26] for example provides an architecture for flexible interaction with electronic devices. Based on OSGi to implement a dynamic module system, their approach provides an abstraction layer

[8] `http://osoa.org/`

for device independence. Their architecture has the following non-functional benefits: scalability and ease of administration, flexibility, security, and efficiency. In a similar vein Escoffier *et al.* proposed iPOJO [13], a service component runtime that simplifies the development of OSGi applications. iPOJO has mainly been used in home-automation to implement service-oriented pervasive applications [7]. AutoHome is a middleware that extends the iPOJO component model, to create a framework to host autonomic home applications. Gaïa [27] is a CORBA-based meta-operating system for ubiquitous computing, built on top of a classical operating system aimint at abstracting the heterogeneity and complexity associated with ubiquitous environments. Olympus [25] proposes a high-level DSL to ease the development of Gaïa applications. Cassou *et al.* [9] proposes a generative programming approach to provide programming, execution and simulation support dedicated to the pervasive computing domain. They also demonstrate how abstraction can help to guide and verify the development of pervasive applications. Again, all these approaches rely on a reconfigurable middleware (often OSGi-based), and this restricts their deployment to powerful nodes, e.g. powerful enough to run a Java virtual machine. Our goal is to provide the same level of abstraction to develop pervasive and adaptive applications for powerful nodes and also for resource-constrained devices.

Several approaches have shown the benefits of using Model Driven Engineering (MDE) to design and reconfigure pervasive applications. Model-based approaches such as Matlab [9], Charon [3], UMLh [8], HyRoom [31], Masachio[16], Mechatronic UML [28], HyVisual [12], or SysML [10] propose a model to code development process with verification techniques to design modular embedded systems. However, none of these approaches support dynamic adaptation of a running system without first designing the adaptation at a business level. This, in practice, significantly reduces the number of configurations these approaches can manage. Indeed, these approaches provide no means to manage the combinatorial explosion of the number of configurations typically encountered in CPS: all configurations need to be explicitly designed.

Several approaches have shown the need of dynamic reconfiguration capabilities for embedded systems. Reconfigurable intelligent sensors are now able to confront major challenges in the design of cost-effective, energy-efficient, customizable systems, for example in the domain of health monitoring systems adaptable to individual users [19], or in the domain of operating system kernels [5]. Run-time reconfiguration can be achieved through programmable logic reconfiguration and/or software adaptation. In the first case, reconfigurable System on Chips [30] are a promising solution. To support software adaptation of embedded software, other works reuse a software architecture-based approach to the construction of embedded systems. For example, The Koala model [32], used for embedded software, allows late binding of reusable components with no additional overhead. Think [14] defines a component-based framework to support different mechanisms for dynamic reconfiguration and to select between them at build time, with no changes in operating system and application components. Different from these approaches, Fleurey *et al.* [15] present an approach based on state machines and an adaptation model to derive adaptive firmwares for microcontrollers. The approach relies on automatically enumerating configurations by exploring a set of adaptation rules defined at design time and compiling the resulting state-machine (which merges the business logic and the adaptation logic) into an optimized, yet static, firmware. In [17], Hofig *et al.* highlight the use of models@runtime for resource-constrained devices. They provide a UML state machine interpreter for AVR microcontrollers and compare the performance overhead with static code generation: model@runtime interpretation is adequate for the majority of situations, except when dealing with high-throughput or delay-sensitive data. Influenced by these approaches, Kevoree leverages models@runtime for microcontrollers and proposes new mechanisms to support dynamic reconfigurations and to select between them at runtime.

[9] `www.mathworks.com/products/matlab/`

[10] `http://www.sysml.org/`

7. CONCLUSION AND PERSPECTIVES

This paper presented μ-Kevoree, which pushes dynamicity and elasticity concerns directly into resource-constrained devices, based on the notion of models@runtime. This modeling layer that micro-controllers expose at runtime, enables the efficient and safe reasoning (by other Kevoree nodes: Java or Android) to adapt microcontroller-based nodes. In particular, this paper focused on the challenges met when mapping Kevoree and models@runtime features to low power microcontrollers, and on the required tradeoffs because of the stringent resource constraints.

This new version of Kevoree has been thoroughly evaluated with benchmarks in order to assess its usability in realistic setups. Despite an overhead with respect to static (non adaptive) code, these benchmarks have shown that 75% of the transactional reconfigurations can be performed in less than 250 ms, which is an acceptable value in many case studies. This is definitely faster (by a factor of almost 50) than a full memory rewrite of the firmware. Also, these benchmarks have shown that the time needed to reboot a microcontroller and restore its previous configuration is a linear function of the script size. For example, booting using a 1 kB EEPROM memory takes between 3 to 15 ms, while this memory size is large enough to store the script of 15 successive reconfigurations before needing compaction. Finally, the benchmarks have shown that Kevoree enables the deployment of software component instances in a number greater than the available pin count on the microcontroller. It is therefore possible to bind a software component to each physical device controlled by the microcontroller, and to deploy an extra component to coordinate these components.

In the future, we will improve the reliability of reconfigurations by making the computation of the initial state step transactional (compression of the persistent memory) *e.g.*, and by exploiting a circular rolling buffer on the persistent memory. Our scheduling algorithm is another area for improvement. Based on existing opportunistic garbage collectors (*e.g.* Java) we will leverage the computational cycles not used by hosted components to trigger the compression of the persistent memory in a lazy way, rather than waiting for a "memory full" event. We will also investigate further optimizations to reduce the memory consumption and the energy consumption. Finally we are planning to integrate simple reasoners in microcontrollers driven by μ-Kevoree so that they can operate in a fully autonomous mode, with no need to delegate the reasoning to a larger node.

Acknowledgment

This work has been funded by the EU FP7 projects: FI-PPP ENVIROFI, S-Cube NoE and MODERATES.

8. REFERENCES

[1] The Internet of Things Meets The Internet of People. `http://www.harborresearch.com/_literature_60961/The_Internet_of_Things_Meets_The_Internet_of_People`.

[2] N. Agrawal, V. Prabhakaran, T. Wobber, J. Davis, M. Manasse, and R. Panigrahy. Design tradeoffs for SSD performance. In *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, pages 57–70. USENIX Association, 2008.

[3] R. Alur, R. Grosu, Y. Hur, V. Kumar, and I. Lee. Modular specification of hybrid systems in charon. In *Proceedings of the Third International Workshop on Hybrid Systems: Computation and Control*, HSCC '00, pages 6–19, London, UK, 2000. Springer-Verlag.

[4] F. André, E. Daubert, N. Grégory, B. Morin, and O. Barais. F4plan: An approach to build efficient adaptation plans. In *MobiQuitous*. ACM, 2010.

[5] S. Bagchi. Nano-kernel: a dynamically reconfigurable kernel for WSN. In *1st international conference on MOBILe Wireless MiddleWARE, Operating Systems, and Applications*, MOBILWARE '08, pages 10:1–10:6, ICST, Brussels, Belgium, Belgium, 2007. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[6] G. S. Blair, N. Bencomo, and R. B. France. Models@run.time. *IEEE Computer*, 42(10):22–27, 2009.

[7] J. Bourcier, A. Diaconescu, P. Lalanda, and J. A. McCann. AutoHome: An Autonomic Management Framework for Pervasive Home Applications. *ACM Trans. Auton. Adapt. Syst.*, 6:8:1–8:10, February 2011.

[8] S. Burmester, H. Giese, and O. Oberschelp. Hybrid uml components for the design of complex self-optimizing mechatronic systems. In J. BRAZ, H. ARAÃŽJO, A. VIEIRA, and B. ENCARNAÃĞÃČO, editors, *INFORMATICS IN CONTROL, AUTOMATION AND ROBOTICS I*, pages 281–288. Springer Netherlands, 2006.

[9] D. Cassou, E. Balland, C. Consel, and J. Lawall. Leveraging Software Architectures to Guide and Verify the Development of Sense/Compute/Control Applications. In *33rd International Conference on Software Engineering (ICSE'11)*, pages 431–440, Honolulu, US, 2011. ACM.

[10] E. M. Dashofy, A. van der Hoek, and R. N. Taylor. An infrastructure for the rapid development of XML-based architecture description languages. In *24th International Conference on Software Engineering*, ICSE '02, pages 266–276, New York, NY, USA, 2002. ACM.

[11] P.-C. David, T. Ledoux, M. Léger, and T. Coupaye. FPath and FScript: Language support for navigation and reliable reconfiguration of Fractal architectures. *Annales des Télécommunications*, 64(1-2):45–63, 2009.

[12] J. Eker, J. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Sachs, and Y. Xiong. Taming heterogeneity - the ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, January 2003.

[13] C. Escoffier, R. S. Hall, and P. Lalanda. iPOJO: an Extensible Service-Oriented Component Framework. In *IEEE SCC*, pages 474–481. IEEE Computer Society, 2007.

[14] J.-P. Fassino, J.-B. Stefani, J. L. Lawall, and G. Muller. Think: A Software Framework for Component-based Operating System Kernels. In *General Track of the annual conference on USENIX Annual Technical Conference*, pages 73–86, Berkeley, CA, USA, 2002. USENIX.

[15] F. Fleurey, B. Morin, and A. Solberg. A Model-Driven Approach to Develop Adaptive Firmwares. In *SEAMS'11@ICSE: Workshop on Software Engineering for Adaptive and Self-Managing Systems*, Honolulu, Hawai, USA, 2011.

[16] T. Henzinger. Masaccio: A formal model for embedded components. In J. van Leeuwen, O. Watanabe, M. Hagiya, P. Mosses, and T. Ito, editors, *Theoretical Computer Science: Exploring New Frontiers of Theoretical Informatics*, volume 1872 of *Lecture Notes in Computer Science*, pages 549–563. Springer Berlin / Heidelberg, 2000.

[17] E. Höfig, P. H. Deussen, and I. Schieferdecker. On the performance of UML state machine interpretation at runtime. In *6th international symposium on Software engineering for adaptive and self-managing systems (SEAMS '11)*, pages 118–127, New York, USA, 2011. ACM.

[18] R. Johnson and B. Woolf. The Type Object Pattern, 1997.

[19] E. Jovanov, A. MilenkoviÄĞ, S. Basham, D. Clark, and D. Kelley. Reconfigurable Intelligent Sensors for Health Monitoring: A Case Study of Oximeter sensor. In *26th Annual International Conference of the IEEE Engineering in Medicine and Biology Society*, pages 4759–4762, 2004.

[20] J. O. Kephart and D. M. Chess. The Vision of Autonomic Computing. *Computer*, 36(1):41–50, 2003.

[21] N. Medvidovic and R. N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Trans. Softw. Eng.*, 26:70–93, January 2000.

[22] B. Morin, O. Barais, J.-M. Jézéquel, F. Fleurey, and A. Solberg. Models@ Run.time to Support Dynamic Adaptation. *Computer*, 42(10):44–51, 2009.

[23] B. Morin, O. Barais, G. Nain, and J.-M. Jezequel. Taming Dynamically Adaptive Systems with Models and Aspects. In *ICSE'09: 31st International Conference on Software Engineering*, Vancouver, Canada, May 2009.

[24] E. D. Nitto, C. Ghezzi, A. Metzger, M. P. Papazoglou, and K. Pohl. A journey to highly dynamic, self-adaptive service-based applications. *Autom. Softw. Eng.*, 15(3-4):313–341, 2008.

[25] A. Ranganathan, S. Chetan, J. Al-Muhtadi, R. H. Campbell, and M. D. Mickunas. Olympus: A High-Level Programming Model for Pervasive Computing Environments. In *Third IEEE International Conference on Pervasive Computing and Communications*, pages 7–16, Washington, DC, USA, 2005. IEEE Computer Society.

[26] J. S. Rellermeyer, O. Riva, and G. Alonso. AlfredO: an architecture for flexible interaction with electronic devices. In *9th ACM/IFIP/USENIX International Conference on Middleware*, Middleware '08, pages 22–41, New York, NY, USA, 2008. Springer-Verlag New York, Inc.

[27] M. Román, C. Hess, R. Cerqueira, A. Ranganathan, R. H. Campbell, and K. Nahrstedt. A Middleware Infrastructure for Active Spaces. *IEEE Pervasive Computing*, 1:74–83, October 2002.

[28] W. Schäfer and H. Wehrheim. Graph transformations and model-driven engineering. chapter Model-driven development with Mechatronic UML, pages 533–554. Springer-Verlag, Berlin, Heidelberg, 2010.

[29] L. Seinturier, P. Merle, R. Rouvoy, D. Romero, V. Schiavoni, and J.-B. Stefani. A Component-Based Middleware Platform for Reconfigurable Service-Oriented Architectures. *Software: Practice and Experience*, 2011.

[30] H. Singh, M.-H. Lee, G. Lu, N. Bagherzadeh, F. J. Kurdahi, and E. M. C. Filho. Morphosys: An integrated reconfigurable system for data-parallel and computation-intensive applications. *IEEE Trans. Comput.*, 49:465–481, May 2000.

[31] T. Stauner, A. Pretschner, and I. Péter. Approaching a discrete-continuous uml: Tool support and formalization. In *Workshop of the pUML-Group on Practical UML-Based Rigorous Development Methods - Countering or Integrating the eXtremists*, pages 242–257. GI, 2001.

[32] R. van Ommering, F. van der Linden, J. Kramer, and J. Magee. The Koala Component Model for Consumer Electronics Software. *Computer*, 33(3):78–85, 2000.

Using Feature Model to build Model Transformation Chains

Vincent Aranega[1], Anne Etien[1], and Sebastien Mosser[2]

[1] LIFL CNRS UMR 8022 Université Lille 1 - France, firstname.lastname@lifl.fr
[2] SINTEF IKT, Norway sebastien.mosser@sintef.no

Abstract. Model transformations are intrinsically related to model-driven engineering. According to the increasing size of standardised meta-model, large transformations needs to be developed to cover them. Several approaches promote to use separation of concerns in this context, that is, the definition of small transformations in order to master their complexity. Unfortunately, the decomposition of transformations into smaller ones raises new issues: organizing the increasing number of transformations and ensuring their composition (*i.e.* the *chaining*). In this paper, we propose to use *feature models* to classify model transformations dedicated to a given business domain. Based on this feature models, automated techniques are used to support the designer, according to two axis: *(i)* the definition of a valid set of model transformations and *(ii)* the generation of an executable chain of model transformation that accurately implement designer's intention. This approach is validated on Gaspard2, a tool dedicated to the design of embedded system.

1 Introduction

Model-Driven Engineering (MDE) advocates the principle of *separation of concerns*, through the extensive use of models in all the steps of the software development cycle [12, 18]. In this context, model transformations are used to achieve *integration of concerns* [14, 17, 3]. Considering the intrinsic complexity of the meta-models in use (*e.g.*, UML 2.x and its profiles), large model transformations are developed. Examples have been published of transformations involving tens of thousands lines of code. Such transformations have substantial drawbacks [15], including reduced opportunities for reuse, reduced scalability, poor separation of concerns, limited learnability, and undesirable sensitivity to changes. The separation of concerns paradigm advocates the decomposition of a complex system (*e.g.*, architectures, object-oriented models) into smaller artefacts. Thus, exactly as other artefacts, it is desirable to *decompose* transformations [20]. Other researches have also argued that focusing on such an engineering of transformations improves the uptake of MDE [22]. It is then essential to support the systematic definition of small model transformations with a unique intention [5], to improve scalability, maintainability and reusability of transformations. Such an approach leads to the definition of a family of transformations associated to a given domain that jointly enable to generate system from a business domain.

The existence of small transformations raises two new issues. First, the chain designer (called *End User* in the remainder of the paper) is in presence of a family of model transformations, that need to be organised. Secondly, the reification of the dependencies that exists between elements of this family becomes critical. As model transformations cannot be chained anyhow, dependencies, that lead to*valid* transformation chain, must be captured. One way to automate this development process is to use a *Software Product Line* (SPL) approach. In a SPL, multiple products are *derived* by combining a set of different core assets. One of the most important challenges of SPL engineering concerns variability management, *i.e.*, how to describe, manage and implement the commonalities and variabilities existing among the members of the same family of products. A well-known approach to variability modeling is by means of *Feature Diagrams* (FD) introduced as part of *Feature Oriented Domain Analysis* [9] back in 1990.

Our contribution in this paper is to accurately combine model transformations and SPL to support the *End User* while developing transformation-based applications. Business experts' knowledge is reified in a FD to accurately organise the different transformations according to their intentions. Then, automated code analysis techniques are used to accurately generate constraints between these transformations[1], reified in the feature model as *requirements* between features. Thus, it is possible for *End users* to use the FD to accurately define their own products, that is, a valid subset of transformations that matches their intentions. Product derivation mechanisms are then used to automatically generate the model transformation chain that implements what the *End User* asked for. The approach is validated using Gaspard2, a transformation-based tool that supports the modelling of embedded systems.

The remainder of this paper is organized as the following. In section 2, we motivate this work by exposing the different challenges that need to be addressed in this domain. Then, section 3 describes the approach we propose to tackle these challenges. Section 4 validates the approach by applying it to the Gaspard2 case study. Finally, section 5 discusses the related works and section 6 concludes this paper by exposing some research perspectives.

2 Motivation

Model driven engineering relies on the principle of separation of concerns to enhance reusability, variability and flexibility and on the abstraction mechanism to enable easier verifications. Gaspard2 [8], a co-design environment dedicated to high performance embedded systems based on massively regular parallelism has been designed using MDE technologies. Thus it enables the generation of VHDL, SystemC, OpenMP or Lustre code from a UML model enhanced with the *Modelling and Analysis of Real Time Embedded systems* (MARTE) profile. Each language is targeted using a chain composed of three to five dedicated

[1] Informally, a transformation τ requires a transformation τ' if the model elements handled by τ are produced by τ'.

transformations. These large transformation (up to 1500 lines of codes) were not reusable and hardly maintainable even by their own developers.

To introduce flexibility and reusability, the Gaspard2 environment has been re-engineered to rely on smaller transformations. Each transformation has a single intention such as memory management or scheduling and corresponds to 150 lines of code in average. Finally, 19 transformations including 4 *model to text* (M2T) transformations, and thus 15 *model to model* (M2M) transformations were defined. The number of chains that can be constructed from them is humongous. Let $T = \{\tau_1, \ldots, \tau_n\}$ a set of model to model transformations, and $M = \{\mu_1, \ldots, \mu_m\}$ a set of model to text transformations. We denote as $N_{T \cup M}$ the number of chains available in this context. The number of potential model to model chains is equal to the number of sequences without repetition that involve elements defined in T (denoted as $P(k, n)$). Secondly, There is $(m+1)$ potential targets for the previously defined sub-chain (as a transformation chain may not generate text). Finally, it is also possible to only generate text without involving other model transformation (thus, m chains).

$$N_{T \cup M} = m + (m+1) \sum_{k=1}^{n} P(k, n), \qquad P(k, n) = \frac{n!}{(n-k)!}$$

$N_{T \cup P}$ is hardly computable generically. Nevertheless, a sub-optimal approximation is to consider $N_{T \cup P}$ bigger than $(m+1)$ times the highest term of the sum $P(k, n)$ (*i.e.*, $P(n, n)$, that in our case is equals to $5 \times 15!$).

$$N_{T \cup P} \gg (m+1) \times P(n, n) = (m+1) \times n!,\ n = 15, m = 4, N_{T \cup P} \gg 6,5 \times 10^{12}$$

But only a few chains make sense! It becomes crucial to help the designer to built chains. Thus, the definition of transformation libraries raises new issues such as *(i)* the representation of the transformations highlighting their purpose and the relationships between them; *(ii)* their appropriate selection according to the characteristics of the expected targeted system and *(iii)* their composition in a valid order.

Traditionally, transformations are represented in chains or with their metamodels. Such representations are not adapted to the description of transformation libraries. In preparation for chaining the transformations, it seems indispensable to specify their purpose (*i.e.*, what they handle), in addition to their associated metamodels. To generate systems with their own characteristics (*e.g.*, management of distributed versus shared memory, optimised vs simple scheduling), transformations have to be consequently selected. However, a selected transformation may require others exactly as when installing a new software that requires non yet installed libraries. Thus the *End User* has to select the transformations not only according to the characteristics of the resulting system she would like, but also to the relationships between the transformations. Manually performed, this selection may be tedious and error prone. From the selected transformations, several chains can be built. Transformations can not be chained no matter how; some constraints must be fulfilled [7, 11]. If it is often

simple to identify the first transformation of the chain (depending on the input metamodels) and the last one (that is a model to text transformation, if code has to be generated), establishing a valid order between the other selected transformations may be difficult. Indeed, existing approaches check if the proposed order is valid, but do not automatically provide a valid one.

In order to support the *End User* in the design of transformation chains, the following challenges have to be addressed:

C_1 Propose to the *End User* a library in which each transformation can be easily identified according to the characteristics of the expected final system (Section 3.1).

C_2 Help the *End User* to select transformations while automatically taking into account the relationships between transformations (Section 3.2).

C_3 Automatically derive the transformation chain from the characteristics selected by the *End User* (Section 3.3).

3 Solution

To tackle the aforementioned challenges, we propose a methodology and tools based on feature-based approach to automatically generate accurate model transformation chains as depicted in Figure 1.

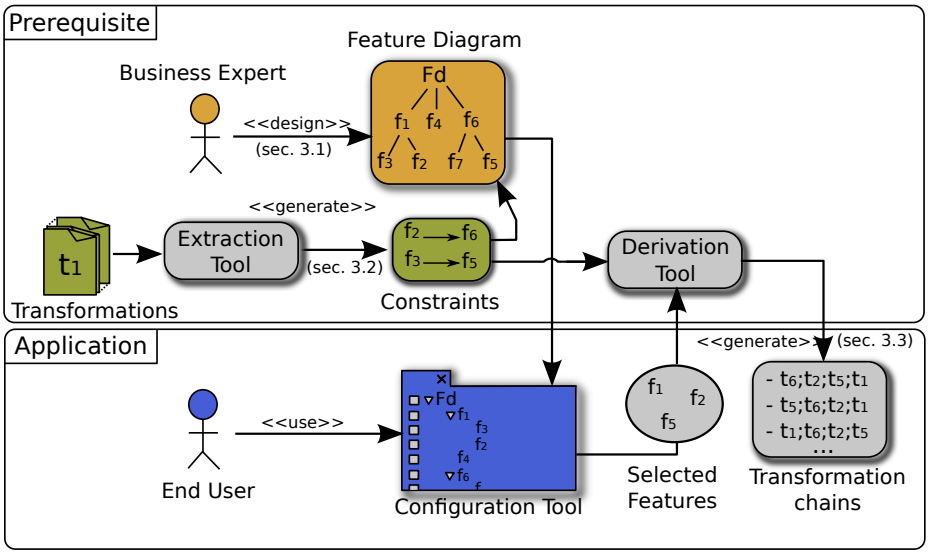


Fig. 1. Approach Process Overview

This approach relies on three pillars: *(i)* the classification of the available transformations as a *Feature Diagram* (FD) produced by the *Business Expert*,

(ii) the reification of requirement relationships between transformation (directly generate from the *Transformations* set by the *Extraction Tool*) and *(iii)* the automated generation of transformation chains for a given product (using our *Derivation Tool*) from features selected by the *End User*.

The FD is designed once for all by the *Business Expert* as a prerequisite. It is nevertheless possible to modify it when new transformations and thus new features are available. The requirement relationships are expressed between the features and automatically computed from the transformation codes by the *Extraction Tool* we provide. The extracted relations enable to derive other features (and then the associated transformation) from the ones selected by the *End User* using a *Configuration Tool* (*e.g.*, FeatureIDE[2]). The requirement relationships are also used by our *Derivation Tool* to order the selected features in order to design valid chains.

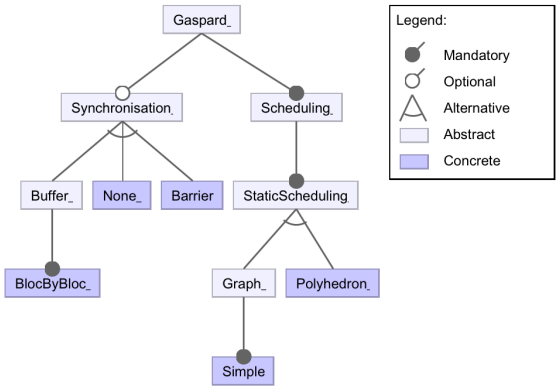
3.1 Structuring the Transformation Set as a Feature Diagram

As a transformation is used to support a given intention according to a business domain, a set of transformations implicitly model the variability of the different intentions associated to a domain. FD where defined to model such a variability, so it is natural to use this modelling approach to support this activity. We represent in Figure 2 an excerpt of the complete FD associated to Gaspard2. Using FD, features (represented as nodes) are classified among others according to constraints such as exclusiveness or optionality. Model transformation are bound to features as assets. A feature f holds a link to the actual model transformation to be used to implement the intention captured by f at run-time.

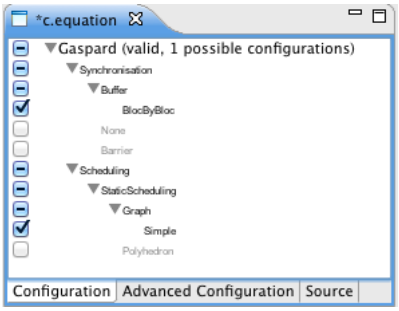
For example, in Figure 2(a), the FD models that a given product *must* contain a `Scheduling` feature, and *may* contain a `Synchronisation` feature. The features `Graph` and `Polyhedron` are exclusive, *i.e.*, the use of one in a given product implies that the other cannot be used in this particular product. We call a product a set of features that respect the constraints modelled in the FD. For example, Figures 2(b) and 2(c) represents two products among the eight valid *w.r.t.* the modelled FD. The first one (Figure 2(b)) considers a system synchronized using a `BlocByBloc` method, and scheduled with a simple `Graph`. The second product (Figure 2(c)) considers a system synchronized with a `Barrier` method, and scheduled with a `Polyhedron` approach. In our context, features reify model transformation: the actual implementation of the transformation is bound as an asset of the associated feature node. Thus, considering a given product, it is possible to automatically infer the set of transformations involved in the transformation chain that supports it.

Key Points. The role of the FD is to capture the business knowledge associated to a given set of transformations. It actually transforms a flat set of transformations into an organised family of products.This classification is done by the *Business Expert*, that is, someone who deeply knows the different transformations, their

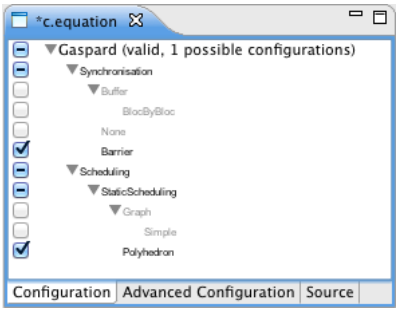
[2] http://wwwiti.cs.uni-magdeburg.de/iti_db/research/featureide/



(a) Gaspard2 feature diagram (excerpt)



(b) $p = \{BlocByBloc, Simple, \dots\}$



(c) $p' = \{Barrier, Polyhedron, \dots\}$

Fig. 2. Gaspard2: Feature diagram and associated products (using FeatureIDE)

underlying intentions, as well as the artifact they are handling. The key idea here is that this work is done once by the *Business Expert*, and capitalised in the FD. Without the use of a FD to support such a classification, it would be up to the *End Users* to guess how the different transformations cope with each others before assembling them.

3.2 Recovering Require Relationship from Transformations

On top of constraints expressing the mandatory/optional character of the features as well as the and/or relationships between them, require relationships can also be captured in FD. They enable to automatically deduced other features from the selected ones, independently of the tree structure of the FD. Require relationships can be determined manually by the *Business Expert*. However, when the number of features is huge, omission can happen leading to erroneous products determination. Therefore, we provide an automatic analysis of the transformations to recover the require relationships between the features associated

to them. In our case, feature requirements is a bijection of transformation requirements: a requirement between two features f and f' (denoted as a logical implication, *i.e.*, $f \Rightarrow f'$) means that the transformation bound to f requires the transformation bound to f'. The following question is raised: "*When does a require relationship between two transformations exists?*". In fact, it relies on the element type production and consumption. For two transformations τ and τ', if τ' consumes types created by τ, then it implies that a require relationship exists between τ and τ', denoted as $\tau' \rightarrow \tau$ (for τ' requires τ). For each transformation, it is thus mandatory to automatically determine the element types it produces and it consumes to provide an automatic require relationships determination.

This automatic analysis relies on the different actions performed on element types by a transformation. Four actions are classically performed by transformations: *reading*, *creating*, *deleting* and *modifying*. This analysis does not rely on transformation execution but on static code analysis. Thus an element of the input or the output metamodel of a transformation is considered read if the presence of one on its instance enables the application of a transformation rule. An element is considered created, if at least one of its instance can be created by the transformation, and so on. Thus, τ' requires τ if τ' reads some elements created by τ [6]. As we stated in the previous section, for a feature f from the FD, it exists at most one transformation τ. So, considering two features f, f' in the FD and two transformations τ, τ' mapped to f, respectively f', if $\tau \rightarrow \tau'$, it implies that $f \Rightarrow f'$.

Key Points. The proposed generation of the require relationships relies on a static analysis of the transformation codes. Once the FD designed and the constraints generated, the *End User* can use a *Configuration Tool* to select the features she wants for her transformation chain. The *Configuration Tool* is parametrised by the feature diagram and the generated constraints. Thus, by taking into account the generated constraints, during the feature selection, the *Configuration Tool* can either invalidate features or add required features according to the ones already selected by the *End User*. The automatic characteristic of the generation enables a certain evolutivity of the FD.

3.3 Generating Transformation Chains

Based on the two previous parts of the contribution, it is now possible to *(i)* consider a set of model transformations as a product family and *(ii)* automatically infer the requirement relationships that exists inside the product family. These two contributions act at the level of the FD. According to the global process, the selected features are then passed to a *Derivation Tool* which uses the generated constraints to propose transformation chains from the selected features.

We consider now a given product $p = \{f_1, \ldots, f_n\}$, *i.e.*, a subset of features that satisfies the constraints modelled in the FD. As stated in Section 3.1, model transformations are bound to features. It is then possible to obtain the set of model transformations associated to p (denoted as T_p) by mapping each feature to its associated transformation: $T_p = \{\tau_1, \ldots, \tau_m\}$. As some features are only

used to structure the FD and are not related to any concrete transformations, it should be noted that the cardinality of T_p may be lesser than the cardinality of p. But this set of transformations is not sufficient to properly derive a concrete transformation chain from a given product. The requirement constraints identified in Section 3.2 must be taken into account. Considering two features f and f', if the requirement $f \Rightarrow f'$ exists, then the transformation τ' mapped to f' must be executed before the transformation τ mapped to f. As a consequence, the analysis of the set of requirement constraints leads to the identification of sequences of model transformations. Two situations can be encountered. If the requirement constraints implement a total order on the set of transformations, only one sequence will be identified, *i.e.*, the proper transformation chain to be executed to support the intentions captured by this product. But if the requirement constraints implement a partial order, only *partial sequences* can be identified automatically. But as there is no requirement between these different sub-sequences, their order is not important. Consequently several valid chains are generated.

Key Points. A concrete chain of model transformations is automatically derived, through the FD, from the transformation set selected by the *End User*. First, the knowledge of a *Business Expert* is captured in the FD, and then an automatic static analysis is used to properly extract technical constraints from the implementation of the transformations. Finally, it is possible to automatically derive the chain, through the systematic exploration of the identified constraints. As a consequence,the generation of the concrete model transformation chain is automated, and the *End User* does not require any knowledge of model transformation from a technical point of view.

4 Validation: The Gaspard2 Case Study

Gaspard2 is a co-design environment dedicated to high performance embedded systems based on massively regular parallelism. From high level specifications, it automatically generates code for high performance computing, hardware-software co-simulation, functional verification or hardware synthesis using model transformations. Such generations are complex and require intermediary steps, *e.g.*, the explicit mapping of application tasks onto processing units, the mapping of the data onto memories or the scheduling of the tasks. Each transformation has a specific intention and deals with few concepts. Nineteen transformations have been implemented for now but the framework may support even more of them in the upcoming months. It is difficult for a non expert user to easily understand the purpose of each transformation, to select the ones useful to reach the desired platform and finally to order them in order to compose a chain.

In this paper, we used the Familiar tool suite [1] to manipulate feature diagrams. This tool allows us to model FD, and is well integrated in the Eclipse platform. Thus, standard *Configuration Tools* (*e.g.*, FeatureIDE) can be used to allow the *End User* to configure products. But it should be noted that the

approach is not bound to this tool nor to this case study from a theoretical point of view, as described in the previous section.

4.1 Step #1: Capturing Business Expert Knowledge in a FD

Embedded systems designers usually do not master model transformation paradigm and underlying technologies. It is then essential to support them while designing the transformation chains used to generate code from high level specifications. The design of these chains consists in the selection of relevant transformations available in a library and then the computation of a valid order. Selecting a transformation requires to easily distinguish one transformation from another and to quickly identify its intention. In order to help the embedded systems designers we have classified the available transformations based on embedded characteristics using feature model. It is up to the *Business Expert* to find the most appropriate classification method to be used to support the *End User*.

Each transformation of Gaspard2 has a unique intention representing a specific characteristics of the produced systems. The intentions identified are related to low level concerns as memory management. The transformations and their associated intention are listed in Table 1. The Gaspard2 transformation library counts 19 intentions through 15 M2M and 4 M2T transformations. For example, the *scheduling* transformation has the follow intention: it manages a simple scheduling of application tasks on computing units.

From these intentions, the *Business Expert* builds the feature diagram by associating a feature to each intention. Moreover, some features are added in the hierarchy in order to specify the relationship AND/OR/XOR between features. Indeed, as stated in Section 3.1, each feature represents at the most one transformation. The resulting FD, depicted in Figure 3, gathers, in an non exhaustive way, some characteristics that an embedded system produced by Gaspard2 can possess. For example, the `OpenCL` and `OpenMP` features, introduce a scientific computation intention. However, only one of these two features could be selected. Indeed, the target language is either OpenCL, or OpenMP. In the FD, this choice is designed by the introduction of an intermediary abstract node `ScientificComputation` and an alternative between the two features.

The associated tooling provided by the Familiar platform can be used to query the model, as shown in Figure 4. This FD models up to 200 different available configurations (obtained by the Familiar `counting` algorithm). The `configs` command computes all the available products, returning the set of valid products defined by this FD.

4.2 Step #2: Extracting Constraints from the Implementation

This feature model enables the classification and the distinction of the transformations the one from the others. However, in this primary form, it does not gather enough information to build the chains. Indeed, exactly as when installing a new library on a computer, some others may be required and the selection of

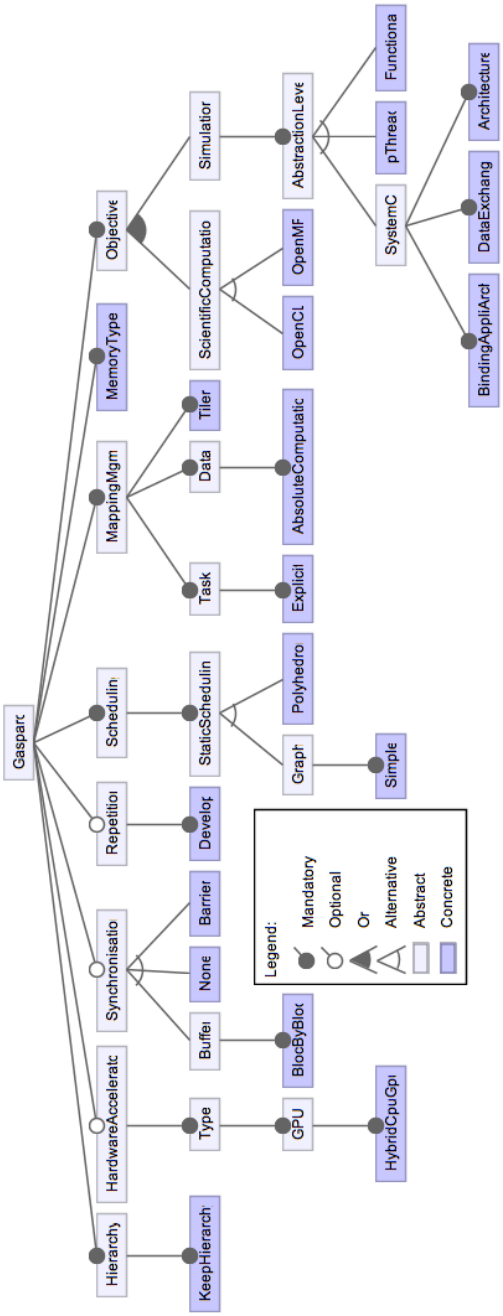


Fig. 3. Feature model associated to Gaspard2.

Transformation	Intention
tiler2task	- Keep repetitions `hierarchy`
gpuApi	- Manage `hybrid GPU-CPU` computing
pThread	- Manage `buffered synchronisation by bloc`
sequentialC	- Generate `sequential` C code
barrier	- Manage `barrier synchronisation`
shape2loop	- `Develop repetitions` in the generated systems
scheduling	- Manage `simple` scheduling
poly_loop	- Manage `polyhedron` optimised `scheduling`
explicitAllocation	- `Explicitly` place tasks no processors
memorymapping	- Manage `absolute` memory `addresses`
tilerMapping	- Manage `tiler` (*i.e.* task distributing data) mapping on computing unit
shared	- Manage the shared `memory type`
openCL	- Generate `OpenCL` code for `scientific computation` purposes
openMP	- Generate `OpenMP` code for `scientific computation` purposes
systemcPA	- `Bind` SystemC `architecture` with SystemC `application`
systemcBind	- Manage SystemC `data exchanges`
systemcStruct	- Manage SystemC `architecture`
pthreadGen	- Generate `pthread` code for simulation purposes
functional	- Introduce `functional` abstraction

Table 1. Gaspard2 transformation set

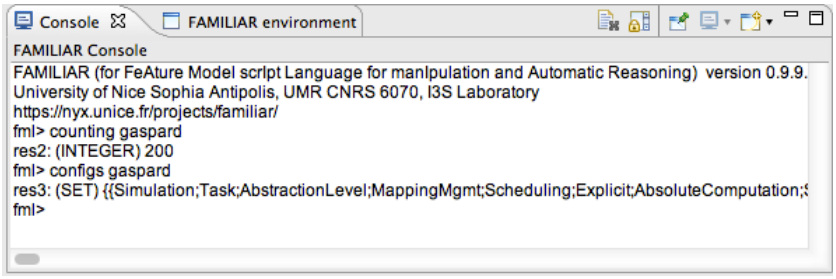


Fig. 4. Using the Familiar shell to interact with the FD.

one transformation may require the selection of others. Such dependencies between transformations have to be captured and the feature model tools enable to take them into account for the product configuration. Thanks to the *Extraction Tool*, the implementation of the available transformations is automatically analysed. The result of this analysis is a set of "require" constraints between the features modelled in the FD. We represent in Listing 1.1 the set of constraints obtained after the execution of the tool. These constraints are generated using the syntax of the Familiar tool, and thus can be automatically integrated in the FD. Contrarily to the initial FD that captures the knowledge of the *Business Expert*, these relations reify the implementation constraints that exist between the

transformations, from a technical point of view. It then ensures that the products configured *w.r.t.* this FD will be valid at both level: *(i)* business domain and *(ii)* technical implementation.

```
1   AbsoluteComputation -> Develop
2   AbsoluteComputation -> KeepHierarchy
3   AbsoluteComputation -> Polyhedron
4   BindingAppliArchi ->
        AbsoluteComputation
5   BindingAppliArchi -> Architecture
6   BindingAppliArchi -> BlocByBloc
7   BindingAppliArchi -> MemoryType
8   BlocByBloc -> AbsoluteComputation
9   BlocByBloc -> Graph
10  BlocByBloc -> KeepHierarchy
11  BlocByBloc -> MemoryType
12  DataExchange -> Architecture
13  DataExchange -> KeepHierarchy
14  Functional -> Graph
15  Graph -> KeepHierarchy
16  Hybrid -> AbsoluteComputation
17  Hybrid -> Graph
18  Hybrid -> KeepHierarchy
19  Hybrid -> MemoryType
20  MemoryType -> KeepHierarchy
21  Simple -> Graph
22  Tiler -> Graph
```

Listing 1.1. Set of requirement constraints.

Considering this set of constraints, the *Configuration Tool* now proposes 37 available products to the *End User* (from 200 at the beginning). This highlights the fact that working with the implementation of the transformation is critical. The technical implementation of the transformations dramatically reduces the initial variability of the domain as it was designed by the *Business Expert*.

Taking into account the "real" features implementations in the FD (*i.e*, the transformations code in our context) through this set of automatically computed constraints also leads to interesting situations that help the *Business Expert*. We consider here the feature `Repetition`, defined as optional by the *Business Expert* (see Figure 3). The generated set of constraints identifies a requirement between the feature `AbsoluteComputation` and the feature `Develop` (line 1 in Listing 1.1). However, `AbsoluteComputation` is mandatory, and selecting `Develop` implies to select `Repetition`. Thus, the `Repetition` feature is automatically identified by the tool suite as a *false optional* feature, that is, a feature modelled as optional but enforced as mandatory by a requirement constraints. In this case, it helped the *Business Expert* to identify a missing artifact in the FD: it should also contain an alternative implementation for `Repetition` instead of only defining the `Develop` approach.

4.3 Step #3: Deriving Transformation Chains

Based on the FD enhanced with the implementation constraints, we can now ensure that the products configured by the *End User* through the configuration tool are valid. The final step is to use a derivation tool that properly builds the transformation chains associated to a given product. We consider here one of the 37 products available according to this FD, denoted as p corresponding for example to the set of the features selected by the *End User*. The first step is to translate p into T_p, that is, the set of transformations involved by this product. It should be noted that $|p| > |T_p|$, as several features are only used to structure the FD and consequently are not bound to concrete transformations. For example,

for the following product, corresponds the associated T_p:

$$\begin{aligned} p = \{&Gaspard, MemoryType, Polyhedron, Data, Barrier, MappingMgmt,\\ &KeepHierarchy, Hierarchy, Tiler, Develop, StaticScheduling,\\ &AbsoluteComputation, Task, Explicit, ScientificComputation,\\ &Scheduling, Objective, Repetition, Synchronisation, OpenMP\}\\ T_p = \{&explicitAllocation, memMapping, openMP, poly\_loop,\\ &shape2loop, tilerMapping\} \end{aligned}$$

The second step is to map the constraints reified between features as a partial order among the transformations. The requirements involved in p are the following:

Feature Requirement $\rightsquigarrow$ **Transformation Ordering**
$AbsoluteComputation \rightarrow Develop \rightsquigarrow memMapping \rightarrow shape2loop$
$AbsoluteComputation \rightarrow Polyhedron \rightsquigarrow memMapping \rightarrow poly\_loop$
$MemoryType \rightarrow KeepHierarchy \rightsquigarrow memMapping \rightarrow tiler2task$

Based on this partial order, it is possible to compute[3] the following sets of "independent" sub-chains involved in this product, as a chain template, that is, a partition of the transformation set taking into account the partial order:

$$tpl_p = [\ [openMP], [explicitAllocation], \quad (1)$$
$$[memMapping, [shape2loop, poly\_loop, tiler2task]\ ] \quad (2)$$

Among the computed sub-chains, the *openMP* transformation is a "model to text" transformation and will always be executed as the last transformation of the chain. In line 2, the partial order indicates that the *memMapping* transformation must be preceded by the 3 transformations listed, without specifying any order between them. Thus, there is up to 6 ways to combine these transformations according to this constraint. As the *explicitAllocation* transformation can be executed independently of these sub-chains, it can be executed before or after the previously described sub-chains. As a consequence, up to 12 chains can be obtained from this product. Following the sub-chains computed by our derivation tool, a valid transformation chain could be:

$$\begin{aligned} &explicitAllocatlion \rightarrow tiler2task \rightarrow \cdots\\ &\qquad\qquad \cdots poly\_loop \rightarrow shape2loop \rightarrow memMapping \rightarrow openMP \end{aligned}$$

Without any lead, the *End User* has only one constraint: the model to text transformation must be the last of the chain. From the product p and its associated set of transformations T_p, it means that the *End User* has the choice to organise 5 transformations. Thus, she has $P(5,5) = 120$ choices to organise the

[3] We used a set of logical predicates implemented using he Prolog language to implement the *Derivation Tool*.

model to model transformations. Among the 120 chaining possibilities, many are not valid because the require relationships are not considered. So, without any indication, the *End User* has to choose from 120, potentially non valid, chains, whereas with our methodology, the choice is reduced to 12 valid chains only.

To sum up, our methodology and the associated tool have allowed the *End User* (without any knowledge about transformations) to easily build chains. She has selected transformations based on embedded system features *i.e.* using terms she is familiar with. Finally, she has to choose among 12 valid chains whereas initially she was confronted to a huge number of possible chains that she has to build by scrutinizing the transformation code.

5 State of the Art

In order to enhance the reusability of transformations, several authors promote the decomposition of transformations into smaller ones. However transformations have then to be chained. Vanhooff and al. propose an approach based on the explicit and manual identification of the required and provided concepts by the chain developer for example using a profile in order to later build the chain [21]. Our approach relies on the feature model to compose the transformations.

Several approaches have been proposed to build chains. Transformations are considered as functions to compose if their domains are compliant [13] or UML activity that can be chained using different operators: composition, conditional composition, parallel composition and loop [16]. However, in both cases, the transformation chain has to be manually specify by the designer, without any specific help. In the latter case, they are executed using the provided model transformation orchestration tool. Our approach could be used upstream to identify the useful transformations and to compose them.

Transformation chaining relies on constraints that can automatically identified, for example using the distinction between concepts copied and those mutated [4]. This approach only deals with endogenous transformations (even if they suggest that an extension to heterogeneous transformations is possible). With the "require" constraints, we have extended this approach to heterogenous transformations.

Several approaches propose to deal with the complexity of large systems with a feature-based approach. For example, feature models were accurately used to model the intrinsic variability of the Linux Kernel [10], and support end-user during the kernel configuration task. The approach proposed in this paper follow the same idea, that is, the use of feature modelling to leverage a highly variable systems into an entity configurable by the end-user.

Being able to extract the features from the implementation is a challenge [2]. The most difficult part is the extraction of the feature hierarchy from the "flattened" implementation [19]. Inferring such a hierarchy relies on domain heuristics that rank the possible hierarchy, and the final assessment of these ranks by a domain expert. In this paper, we do not consider the automatic extraction of

the features from the transformation set, and only rely on the business expert to properly model the feature diagram. Being able to support the business expert during this task is an interesting perspective of this work.

Feature models are also used to support the reverse engineering of large scale systems [1]. For example, the FraSCAti platform (an open source implementation of the SCA standard) was accurately reverse-engineered to support its assessment. Based on a dedicated tool that extracts the architecture from the implementation, the authors confronts the automatically extracted feature model with the one defined by the business expert. This approach complements our approach, as we also rely on a tool that automatically infers feature information from the actual implementation of the system (in our case requirements between features). But instead of assessing the model defined by the business expert, we focused on its enrichment, by merging the set of automatically identified information in this feature model. We were able to identify several situations where the actual system was not "as variable" as the business expert thought.

6 Conclusions & Perspectives

In order to be reusable and maintainable, model transformation are written according to a single intention and complex transformation are built as the chaining of smallest ones. In this paper, we proposed an approach based on FD to support the design of model transformation chains. Based on a classification of the transformation made by a *Business Expert*, This approach allows an *End User* to build such chains, without any prior knowledge of model transformation technologies. The implementation of the transformations is also automatically taken into account to ensure that the built chains are valid from a run-time point of view. From an implementation point of view, the approach is independent of any tools and can be easily coupled to existing approaches (*e.g.*, FeatureIDE, Familiar). The approach was validated on the Gaspard2 case study, and we are currently pursuing another validation study in the domain of website engineering.

The resulting chains are valid according to a type based approach [7]. However, two transformations that can be chained into both orders from a syntactic perspective are not obviously commutable from a business point of view: the execution of the two successive transformations on whatever models may not always lead to the same result. A perspective of this work is to enhance the expressiveness of the requirement detection mechanisms to address this issue.Another perspective concerns the FD refinement. Indeed, the FD being manually designed by the *Business Expert*, some constraints between features may have been omitted. The automatic requirement relationships extraction could be a first help to highlight a badly / incompletely designed FD. To help the *Business Expert* in the definition or the refinement of the FD, we plan to automatically extract features from the documentation written by the transformation developers.

References

1. Mathieu Acher, Anthony Cleve, Philippe Collet, Philippe Merle, Laurence Duchien, and Philippe Lahire. Reverse engineering architectural feature models. In Ivica Crnkovic, Volker Gruhn, and Matthias Book, editors, *ECSA*, volume 6903 of *Lecture Notes in Computer Science*, pages 220–235. Springer, 2011.
2. Mathieu Acher, Anthony Cleve, Gilles Perrouin, Patrick Heymans, Charles Vanbeneden, Philippe Collet, and Philippe Lahire. On extracting feature models from product descriptions. In Ulrich W. Eisenecker, Sven Apel, and Stefania Gnesi, editors, *VaMoS*, pages 45–54. ACM, 2012.
3. Jean Bézivin and Olivier Gerbé. Towards a Precise Definition of the OMG/MDA Framework. In *16th IEEE International Conference on Automated Software Engineering (ASE 2001), San Diego, CA, USA*, pages 273–280, 2001.
4. Raphael Chenouard and Frédéric Jouault. Automatically Discovering Hidden Transformation Chaining Constraints. In Andy Schurr and Bran Selic, editors, *Model Driven Engineering Languages and Systems*, volume 5795 of *Lecture Notes in Computer Science*, pages 92–106. Springer Berlin / Heidelberg, 2009.
5. Jim Cordy. Eating our own Dog Food: DSLs for Generative and Transformational Engineering. In *GPCE*, 2009.
6. A. Etien, V. Aranega, X. Blanc, and R. Paige. Chaining Model Transformations. In *Submitted to ECMFA conference*, 2012.
7. A. Etien, A. Muller, T. Legrand, and X. Blanc. Combining Independent Model Transformations. In *Proceedings of the ACM SAC, Software Engineering Track*, pages pp. 2239–2345, 2010.
8. A. Gamatié, S. Le Beux, É. Piel, R. Ben Atitallah, A. Etien, P. Marquet, and J.-L. Dekeyser. A Model Driven Design Framework for Massively Parallel Embedded Systems. *ACM Transactions on Embedded Computing Systems*, 10(4), 2011.
9. Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. Feature-Oriented Domain Analysis (FODA) - Feasibility Study. Technical report, The Software Engineering Institute, 1990.
10. Rafael Lotufo, Steven She, Thorsten Berger, Krzysztof Czarnecki, and Andrzej Wasowski. Evolution of the linux kernel variability model. In Jan Bosch and Jaejoon Lee, editors, *SPLC*, volume 6287 of *Lecture Notes in Computer Science*, pages 136–150. Springer, 2010.
11. T. Mens, G. Taentzer, and O. Runge. Detecting Structural Refactoring Conflicts Using Critical Pair Analysis. *Electronic Notes in Theoretical Computer Science*, 127(3):113–128, April 2005.
12. J. Miller and J. Mukerji. MDA Guide Version 1.0.1. Technical report, Object Management Group (OMG), 2003.
13. Jon Oldevik. Transformation Composition Modelling Framework. In *Proceedings of the Distributed Applications and Interoperable Systems Conference*, volume 3543 of *Lecture Notes in Computer Science*, pages 108–114. Springer, 2005.
14. Harold Ossher, William Harrison, and Peri Tarr. Software engineering tools and environments: a roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pages 261–277, New York, NY, USA, 2000. ACM.
15. Jens Pilgrim, Bert Vanhooff, Immo Schulz-Gerlach, and Yolande Berbers. Constructing and Visualizing Transformation Chains. In *Proceedings of the European conference on Model Driven Architecture*, pages 17–32, 2008.
16. José E. Rivera, Daniel Ruiz-Gonzalez, Fernando Lopez-Romero, José Bautista, and Antonio Vallecillo. Orchestrating ATL Model Transformations. In *Proc. of MtATL 2009*, pages 34–46, Nantes, France, July 2009.

17. Douglas C. Schmidt. Guest Editor's Introduction: Model-Driven Engineering. *IEEE Computer*, 39(2):25–31, 2006.
18. Bran Selic. The Pragmatics of Model-Driven Development. *IEEE Software*, 20(5):19–25, 2003.
19. Steven She, Rafael Lotufo, Thorsten Berger, Andrzej Wasowski, and Krzysztof Czarnecki. Reverse engineering feature models. In Richard N. Taylor, Harald Gall, and Nenad Medvidovic, editors, *ICSE*, pages 461–470. ACM, 2011.
20. Bert Vanhooff, Dhouha Ayed, and Yolande Berbers. A Framework for Transformation Chain Development Processes. In *Proceedings of the ECMDA Composition of Model Transformations Workshop*, pages pp. 3–8, 2006.
21. Bert Vanhooff and Yolande Berbers. Breaking Up the Transformation Chain. In *Proceedings of the Best Practices for Model-Driven Software Development at OOPSLA 2005*, San Diego, California, USA, 2005.
22. Dennis Wagelaar, Ragnhild Van Der Straeten, and Dirk Deridder. Module Superimposition: a Composition Technique for Rule-based Model Transformation Languages. *Software and Systems Modeling*, 2009. Online First.

Migrating Component-based Web Applications to Web Services: towards considering a "Web Interface as a Service" -Extended Abstract-

Chouki Tibermacine* and Mohamed Lamine Kerdoudi**

*LIRMM, CNRS and Montpellier II University, France
Chouki.Tibermacine@lirmm.fr
and
**Computer Science Department
University of Biskra, Algeria
lamine.kerdoudi@gmail.com

Abstract. Currently, Web applications in industry are built mainly by customizing and assembling reusable Web components. These components embed business logic and presentation views, and are subject to instantiation and deployment on Web servers. The problem with this kind of applications is that after their construction, there is no way to open their provided functionality for extension by third party developers. In this extended abstract, we present a method for transforming component-based Web applications into service-oriented ones. In this way, these applications which were usable only via their provided Web interfaces will be accessible for external developers. To a certain extent, we develop here a new concept of a "Web interface as a Service".

1 Introduction: Context and Problem Statement

Nowadays, industrial Web applications are built mainly by assembling reusable Web components. These components are first customized to meet the requirements of the built application, then instantiated and assembled with other component instances. This application is then deployed in a Web server in order to be tested and validated. Finally, it is put into production by deploying it in a Web/application server accessible for final users. Once deployed such applications are accessible only via their Web interfaces. There is no way for an external developer to implement an extension to this Web application, by exploiting for example the HTML results returned after querying the application via its Web interface. Existing techniques provided by IDEs and application servers are language-dependent and are not suited for the Web.

In this extended abstract, we present a method (presented in detail in [4]) for transforming component-based Web applications to Web service-oriented ones. The components of Web applications are parsed, and their functionality is extracted and published as operations in Web services. In addition, Web interfaces provided by these applications are transformed into operations that are published also. Navigations between the Web interfaces exposed by the application and collaborations between the published Web services are identified, and orchestrations/choreographies are generated. We implemented this method for Java EE.

In the following section, we detail this transformation method and its implementation. Then, we conclude by presenting some related works, a synthesis of the contribution, and some perspectives.

2 Proposed Approach and its Implementation

The transformation is performed through a semi-automatic process composed of 6 steps:

1. **Operation Extraction:** This process starts by statically analyzing the contents of the individual Web components forming the application to be transformed. All methods in classes and

functions in scripts are copied to be considered as potential operations in the Web services that will be created and published. The identified methods and functions are first made stateless (transform accesses to global variables or attributes into additional parameters, and making the necessary refactorings). In addition, new operations are generated from the Web interfaces exposed by the analyzed components. These operations have as parameters the data entered by users when manipulating the Web interface (data entered in forms, for exemple), and have as output the results returned by the scripts processing the data entered by the users.

2. **Input and Output Message Identification:** In this step, messages are created for the operations extracted in the previous step. The data types are inferred from the texts of the analyzed programs (scripts and classes).
3. **Operation Filtering:** All operations that must not be published in Web services are eliminated in this step. This is a manual task, which needs the knowledge of the developers. Nevertheless, we provide an automatic way of eliminating unwanted recurrent operations. This is done through OCL constraints that developers should specify on a simple metamodel of operations inspired from the UML metamodel. These constraints are automatically checked. The operations kept in this step must satisfy these constraints, if any.
4. **Operation Distribution:** In this step, Web service descriptions are created. Operations tightly coupled are grouped in the same Web services in order to enhance their performance, and operations that are similar (lexically and semantically) are spread-out in different Web services, in order to enhance their reliability.
5. **Composite Web Service Creation:** Navigations between Web interfaces exposed by the components of the application enable to create orchestrations of the previously generated Web services (embedding the operations that have been created from the Web interfaces). In addition, choreographies are generated from collaborations that are identified by analyzing the relationships between the published Web services.
6. **Web Service Deployment and Indexing:** In this final step, assistance is given to developers in order to deploy the generated primitive and composite Web services. In addition, it proposes a tool which automatically extracts keywords from the generated Web service descriptions.

The current implementation of this process parses JEE components. It generates WSDL files, Java code, BPEL processes (for orchestrations) and WS-CDL specifications (for choreographies).

3 Conclusion: Related, Ongoing and Future Work

The main contribution of the work presented in this extended abstract can be summarized as the concretization of the idea of opening for third parties the development of extensions of legacy component-based Web applications by transforming their embedded Web interfaces and functionality into operations published in Web services.

Existing works, such as [1–3], propose interesting methods for transforming existing code into (Web) services. They however neither address the transformation of Web interfaces into services, nor propose the generation of composite services, as done in our work.

Currently, we are experimenting the generation of WS-CDL specifications. In the near future, we plan to validate this work by experimenting and validating our transformation method on real-world component-based Web applications.

References

1. H. Han and T. Tokuda. Wike: A web information/knowledge extraction system for web service generation. In *Proc. of ICWE'08*, pages 354–357. IEEE CS, 2008.
2. R. Lee, A. Harikumar, C.-C. Chiang, H.-S. Yang, H.-K. Kim, and B. Kang. A framework for dynamically converting components to web services. In *Proc. of SERA'05*, 2005.
3. A. Marinho, L. Murta , and C. Werner Extending a Software Component Repository to Provide Services. In *Proc. of ICSR'09.* Falls Church, USA, pp. 258-268, 2009.
4. C. Tibermacine and M. L. Kerdoudi. Migrating Component-based Web Applications to Web Services: towards considering a "Web Interface as a Service". In *Proc. of ICWS'12*, IEEE Computer Society, 2012.

Session du groupe de travail Compilation

Vers une compilation energy-aware

Auteur : Olivier Zendra (Inria Nancy Grand Est)

Résumé :
Dans cet exposé nous ferons un point sur les problématiques énergétiques des nouvelles plateformes matérielles, et leur impact sur la compilation des logiciels.

Règles de Réécriture Multi-Focus, un point de vue orienté compilation

Auteur : Christophe Calvès (LORIA, U. Lorraine)

Résumé :
La réécriture est fréquemment utilisée en compilation afin d'optimiser, de transformer ou générer du code. Nous présentons dans cet exposé des règles de réécriture à multiple focus, capables de parcourir et filtrer, en une seule règle, plusieurs sous-termes d'un sujet. Cet exposé inclut une présentation d'une implémentation en TOM, un compilateur embarquant termes algébriques et règles de réécritures dans de nombreux langages.

Analyses statiques pour la génération de code synchrone

Auteur : Laure Gonnord (LIFL, U. Lille)

Résumé :
Dans cet exposé nous décrivons la compilation des programmes synchrones et l'utilité d'une analyse statique performante pour la génération du code. Nous montrons comment une abstraction booléenne/numérique peut améliorer cette analyse.

Session du groupe de travail FORWAL

Formalismes et Outils pour la Vérification et la Validation

Automates d'arbre avec un nombre fixe de contraintes

P.-C. Héam et V. Hugot et O. Kouchnarenko

FEMTO-ST - CNRS UMR 6174 - INRIA CASSIS

1 Introduction

Les résultats énoncés ici ont été publiés dans [4]. Les automates d'arbre sont une extension naturelle des automates sur les mots et fournissent un modèle simple de calcul sur les arbres [2]. Un arbre n'étant pas une structure purement linéaire, de nombreuses extensions ont été proposées afin de permettre de comparer, lors de l'exécution d'un calcul, l'égalité ou la différences de certains sous-arbres. La plupart des extensions naturelles conduisent à des classes d'automates pour lesquels le problème du vide devient alors indécidable. Dans [3], afin d'étudier le traitement de document structurés comme XML, les automates d'arbres avec contraintes d'égalité et de différence ont été introduits. Pour cette classe, le problème de l'appartenance d'un mot est NP-complet [3], les problèmes de vacuité (problème du vide) et de finitude sont EXPTIME-complet [3,1] et l'inclusion est indécidable [5]. Nous montrerons que sans contrainte d'inégalité et si le nombre de contraintes d'égalité est une constante du problème, alors l'appartenance se décide en temps polynomial. S'il n'y a qu'une contrainte, le problème du vide est lui aussi polynomial. A partir de deux contraintes, il reste EXPTIME-complet.

2 Automates d'arbre avec contraintes d'égalité

Dans la suite, on travail sur un alphabet $\mathcal{F}$ avec arité. Un automate d'arbre est un tuple (Q, Δ, F) où Q est l'ensemble des états, Δ l'ensemble des transitions et F l'ensemble des états finaux. Les transitions sont des règles de la forme $f(q_1, \ldots, q_n) \to q$ où les q_i et q sont des états et $f \in \mathcal{F}$ est d'arité n. Une exécution d'un automate d'arbre sur un arbre t est une application ρ des positions (des nœuds) de t vers q telle que si une position p de t est étiquetée par f d'arité n, alors les positions $p_1, \ldots, p_n$ des fils de f vérifient $f(\rho(p_1), \ldots, \rho(p_n)) \to \rho(p) \in \Delta$. Une exécution ρ est acceptante si l'image de la racine par ρ est un état final. Un arbre est reconnu par un automate d'arbre s'il existe une exécution acceptante sur cet arbre.

Par exemple, si l'on considère que $\mathcal{F} = \{a, b, f\}$ où a et b sont d'arité 0 et f est d'arité 2 et si $Q = \{q_0, q_1, q_f\}$, $\Delta = \{a \to q_0, b \to q_0, f(q_0, q_0) \to q_0, f(q_0, q_0) \to q_1, f(q_1, q_1) \to q_f\}$ et $F = \{q_f\}$, alors l'ensemble des arbres acceptés par (Q, Δ, F) est l'ensemble des arbres ayant un f à la racine. Un automate d'arbre avec contraintes d'égalité (TAGE – *Tree Automata with Global*

Equality constraints) est un automate d'arbre muni d'une relation binaire E sur Q. Un arbre t est reconnu par un TAGE s'il existe une exécution acceptante ρ sur cet arbre vérifiant : pour tout couple (p_1, p_2) de positions de t, si $(\rho(p_1), \rho(p_2)) \in E$ alors les sous-arbres de t enracinés en p_1 et p_2 sont les mêmes. Sur l'exemple, si l'on prend $E = \{(q_1, q_1)\}$, l'ensemble des arbres reconnus sont ceux de la forme $f(t, t)$ où t est un arbre quelconque. Les résultats suivants [1] sont présentés dans [5,3].

Theorem 1. *Pour les automates d'arbres à contraintes d'égalité, le problème du vide est EXPTIME-complet ainsi que celui de la finitude du langage reconnu. Le problème de l'appartenance est NP-complet.*

3 Avec un nombre fixe de contraintes

Le théorème 1 donne des complexités élevées. Comme pour certaines applications le nombre de contraintes est très réduit, nous avons étudié ces problèmes en fonction du nombre de contraintes d'égalité. Un k-TAGE est un TAGE où la relation d'égalité est de cardinal au plus k. Nous avons prouvé les résultats suivants.

Theorem 2. *Pour la classe des 1-TAGE, le problème du vide et le problème de la finitude sont dans P, et ils sont EXPTIME-complets pour les k-TAGE, si $k \geq 2$. Le problème de l'appartenance est dans P pour tous les k-TAGE, quelque soit k fixé.*

Notons que l'algorithme proposé pour l'appartenance est exponentiel par rapport à k, et ne peut donc être utile que pour un faible nombre de contraintes. Dans le futur, nous envisageons d'étendre les résultats en ajoutant des contraintes d'inégalité.

Références

1. L. Barguñó, C. Creus, G. Godoy, F. Jacquemard, and C. Vacher. The emptiness problem for tree automata with global constraints. In *LICS*, pages 263–272. IEEE Computer Society, 2010.
2. H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications, 2007.
3. E. Filiot, J.-M. Talbot, and S. Tison. Tree automata with global constraints. *Int. J. Found. Comput. Sci.*, 21(4) :571–596, 2010.
4. P.-C. Héam, V. Hugot, and O. Kouchnarenko. On positive taged with a bounded number of constraints. In N. Moreira and R. Reis, editors, *CIAA*, volume 7381 of *Lecture Notes in Computer Science*, pages 329–336. Springer, 2012.
5. F. Jacquemard, F. Klay, and C. Vacher. Rigid tree automata and applications. *Inf. Comput.*, 209(3) :486–512, 2011.

1. Des résultats similaires, présentés dans les mêmes articles, existent si l'on ajoute en plus une relation de différence.

Model Checking régulier pour automate d'arbres à treillis

Thomas Genet[1], Tristan Le Gall[2], Axel Legay[1], and Valérie Murat[1]

[1] INRIA/IRISA, Rennes
[2] CEA, LIST, Centre de recherche de Saclay

Le model checking régulier sur termes (Tree Regular Model Checking : $TRMC$) est une famille de techniques permettant d'analyser les systèmes à espace d'états infini dans lequel les états sont représentés par des termes, et les ensembles de termes par des automates d'arbres. Le problème principal du $TRMC$ est de savoir si un ensemble d'états erreur est accessible ou non. Le calcul d'un automate d'arbres représentant (une sur-approximation de) l'ensemble des états accessibles est un problème indécidable. Mais des solutions efficaces basées sur la complétion existent. Malheureusement, les techniques actuelles liées au $TRMC$ ne permettent pas de capturer efficacement à la fois la structure complexe d'un système et certaines de ces caractéristiques. Si on prend par exemple les programmes Java, la structure d'un terme est principalement exploitée pour modéliser la structure d'un état du système. En contrepartie, les entiers présents dans le programme Java doivent être encodés par des entiers de Peano, donc chaque opération algébrique est potentiellement modélisée par une centaine d'applications de règles de réécriture. Nous proposons ici des automates d'arbres à treillis (LTA), une version étendue des automates d'arbres dont les feuilles sont équipés avec des éléments d'un treillis. Les LTA nous permettent de représenter des ensembles possiblement infinis de termes pouvant être interprétés. Ces termes "interprétables" permettent de représenter efficacement des domaines complexes et leurs opérations associées. Enfin, en tant que contribution principale, nous définissons un nouvel algorithme de complétion permettant de calculer l'ensemble possiblement infini des termes interprétables accessibles en un temps fini.

Les modèles à ensemble d'états infini sont souvent utilisés pour éliminer les hypothèses potentiellement artificielles sur la structure et l'architecture des données analysées, comme par exemple une borne artificielle sur la taille d'une pile ou sur la valeur d'une variable. On trouve, dans la plupart des techniques proposées pour explorer les espaces d'états infinis, une représentation symbolique pouvant représenter de façon finie un ensemble infini d'états. Depuis plusieurs années, avec l'apparition des automate de mots, s'est imposée l'idée qu'une représentation générique sous forme d'automates pour représenter des ensembles d'états pouvait être utilisée dans de nombreux cas. Cette idée s'est étendue au principe plus général de model-checking régulier pour automate d'arbres ($TRMC$). Dans cette technique, les états sont représentés par des *arbres* (ou *termes*), les ensemble d'états (possiblement infinis) par des automates d'arbres, et le comportement du système à vérifier par des règles de réécritures ou encore des transducteurs d'arbres. Contrairement aux approches spécifiques, le $TRMC$ est une méthode générique et suffisamment expressive pour décrire une grande partie des protocoles de communication, des programmes en C avec des structures de données complexes, des programmes multi-threadés, ainsi

que des protocoles cryptographiques. Dans chacune de ses approches, le $TRMC$ est équipé d'un algorithme permettant d'accélérer le calcul d'ensembles potentiellement infinis d'états afin que ce calcul se déroule en un temps fini. Parmi ces algorithmes, nous pourrons citer la complétion par abstraction à l'aide d'un ensemble d'équations, qui calcule les automates successifs obtenus par application des règles de réécriture, puis fusionne, à chaque nouvelle étape, les états équivalents par équation. Ainsi la convergence du calcul est favorisée.

Certains travaux proposent une traduction exacte de la sémantique de la JVM Java aux règles de réécriture et automates d'arbres. Cette traduction permet une analyse des programmes Java avec les model-checkers classiques utilisant le $TRMC$. Une des principales difficultés est de capturer et gérer les deux dimensions infinies présentes dans les programmes Java. En effet, des comportements infinis peuvent venir d'appels en boucle à des méthodes ou création d'objets, ou tout simplement parce que le programme manipule des données infinies comme les variables numériques contenant des entiers. Les multiples comportements infinis peuvent être sur-approximés grâce aux techniques d'accélération citées précédemment comme la complétion par abstraction à l'aide d'équations, mais les entiers et leurs opérations doivent être représentés en arithmétique de Peano : en effet ceci permet de modéliser les entiers sous forme de termes et les opérations sous forme de règles de réécriture. Mais cette représentation a un impact exponentiel sur la taille des automates représentants les ensembles d'états ainsi que sur le processus de calcul. Par exemple, l'addition $x + y$ nécessite l'application de x règles de réécriture.

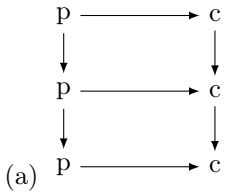
Une solution à ce problème serait de représenter les entiers directement dans les automates d'arbres et systèmes de réécritures, ainsi que leurs opérations, comme faisant partie de l'alphabet utilisé. Dans ce cas, chaque terme possédant un élément de cet alphabet particulier, comme une opération entre deux entiers, doit être interprété et retourner directement le résultat de l'opération sans appliquer aucune règle de réécriture. Notre objectif est donc l'étude d'une nouvelle approche de $TRMC$ intégrant ces nouveaux types de termes "interprétables". Notre première contribution est la définition des automates d'arbres à treillis (Lattice Tree Automata : LTA), une nouvelle classe d'automates d'arbres capable de représenter des ensembles potentiellement infinis de termes interprétables. Intuitivement, les LTA sont des automates d'arbres dont les feuilles peuvent être équipées d'éléments d'un treillis permettant d'abstraire des ensembles possiblement infinis de valeurs. Les noeuds des LTA peuvent être des éléments d'un alphabet "classique" ou représenter des opérations sur les éléments du treillis. Nous proposons également un nouvel algorithme d'accélération pour calculer l'ensemble des états accessibles à partir de LTA, en étendant l'algorithme de complétion classique en considérant des systèmes de réécritures conditionnels. Nous proposons aussi un nouveau type d'équations pour favoriser la convergence de cette nouvelle approche. Enfin, la correction de l'algorithme est prouvée, et cette propriété est garantie grâce à l'existence d'une étape d'évaluation intégrée à l'algorithme de complétion. Finalement, nous parlerons de l'implémentation de cette solution et de comment elle peut grandement améliorer la vérification de programmes Java utilisant les techniques de $TRMC$.

Γ-Pomsets

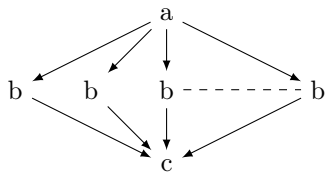
Jean-Michel Couvreur, Mouhamadou Tafsir Sakho

Université d'Orléans,
Laboratoire d'Informatique Fondamentale d'Orléans,
F-45067 ORLEANS Cedex 2, France
{jean-michel.couvreur,mohamadou.sakho}@univ-orleans.fr
http://www.univ-orleans.fr/lifo/

Dans le domaine de la vérification formelle, les comportements d'un système sont en généralement modéliser par des mots finis ou infinis. Chaque lettre représente une action ou les valeurs des propriétés élémentaires de l'état d'un système. On parle de comportements linéaires dans le sens où la suite des événements sont totalement ordonnés. Dès que nous affaiblissons la contrainte d'un ordre total à un ordre partiel, l'ordre induit une relation de causalité sur les événements et deux événements incomparables sont considés comme concurents. Formellement, un comportement d'un système concurrent est un pomset (partial ordre multiset) [5]. Les figures (a) et (b) donnent deux exemples de pomset représentés par un graphe acyclique où des arcs orientés définissent des relations de causalité entre événements. L'exemple (a) modélise le comportement élémentaire d'un système producteur-consommateur, alors que (b) est un modèle série-parallèle représentant un comportement où l'action a est suivi d'un nombre indéterminé d'actions b concurrentes, terminant par une action c.



(a)

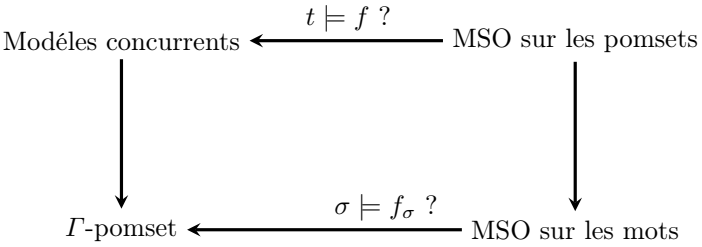


(b)

La vérification de systèmes concurrents pour des formules de logiques temporelles *ordre partiel* usuelles sur un système concurrent est complexe. Il est à noter que le problème de la satisfaction d'une formule est en général indécidable. Il existe cependant des résultats de décidabilités pour la vérification de certaines classes de modèles de la concurrence telles que des expressions régulières de séries-parallèles [3], de MSC (Message Sequence Chart) [4, 2] et de traces de Mazurkiewicz [1]. Dans nos travaux, nous avons retrouvé et généralisé ces résultats en transposant le problème de la vérification de systèmes concurrents à un problème de même nature que ceux de la vérification sur des systèmes non concurents.

L'idée de notre approche est de représenter un pomset par un mot. Ce mot est contruit à partir d'une linéarisation du pomset enrichie de marques modélisant les relations de causalité entre événements. Formellement, nous introduisons la notion de Γ-pomsets comme un mot sur un alphabet $2^\Gamma \times \Sigma \times 2^\Gamma$ où Γ est un ensemble fini de *marques* et Σ est l'alphabet étiquetant les événements des pomsets.

Chaque lettre $\langle pre_i, a_i, post_i\rangle$ d'un Γ-pomsets $\langle pre_0, a_0, post_0\rangle \cdots \langle pre_n, a_n, post_n\rangle$ modélise un événement étiqueté a_i. Nous dirons que l'événement $\langle pre_i, a_i, post_i\rangle$ est plus petit que (ou bien après) l'événement $\langle pre_j, a_j, post_j\rangle$ (avec $i < j$) si il existe une marque dans $post_i \cap pre_j$ n'appartenant à aucun ensemble $post_k$ pour $i < k < j$. Plus précisement, la relation de causalité est obtenu par la fermeture transitive de la relation que nous venons juste de donner. En observant que cette relation de causalité s'exprime par une formule de logique monodique sur les mots, nous déduisons que la vérification d'une formule de logique monadique sur les pomsets est décidable sur les expressions régulières de Γ-pomsets. La figure ci-dessous resume le principe de la méthode de vérification.



Appliquer la méthode à un modèle de la concurence revient à élaborer des traductions en expressions régulières de Γ-pomsets. La traduction de l'exemple (a) utilise deux marques (0 et 1) et donne $(\langle 0, p, 0\rangle \cdot \langle \{0, 1\}, c, 1\rangle)^*$. Par contre la traduction de l'exemple (b) nécessite l'introduction d'événements non observables afin que le nombre de prédécesseurs immédiats de tous événements soit limité à deux. La traduction utilise 3 marques et donne :

$$\langle \emptyset, a, \{0, 1\}\rangle \cdot (\langle 0, b, 2\rangle \cdot \langle \{1, 2\}, \varepsilon, 1\rangle)^* \cdot \langle 1, c, \emptyset\rangle$$

Actuellement, nous savons traiter plusieurs modèles classiques de la concurrence : les séries-parallèles , les MSC, les traces de Mazurkiewicz et les réseaux de Petri sain. Nous pensons que notre approche est bien adaptée pour la conception d'un outil de vérification traitant des logiques temporelles *ordre partiel* intégrant des techniques d'optimisation telles que le calcul symbolique sur les modèles, et les méthodes de réduction *ordre partiel* du graphe d'états.

References

1. Volker Diekert. *The Book of Traces.* World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1995.
2. Jesper G. Henriksen, Madhavan Mukund, K. Narayan Kumar, Milind A. Sohoni, and P. S. Thiagarajan. A theory of regular msc languages. *Information and Computation/information and Control*, 202:1–38, 2005.
3. K. Lodaya and P. Weil. Series-parallel posets: Algebra, automata and languages. In *STACS98, Lecture Nodes in Computer Science*, pages 555–565. Springer, 1998.
4. Remi Morin. Recognizable sets of message sequence charts. In *STACS 2002, LNCS 2030*, pages 523–534. Springer, 2002.
5. Vaughan R. Pratt. Modelling concurrency with partial orders. *International Journal of Parallel Programming*, 15(1):33–71, 1986.

Session de l'action IDM

Ingénierie dirigée par les modèles

Precise Semantics of UML Composite Structures : Overview of an Ongoing OMG Standard

Arnaud Cuccuru

CEA LIST DILS
Point Courrier n 174
91191 Gif-sur-Yvette Cedex, France
arnaud.cuccuru@cea.fr

Résumé The purpose of this presentation is to give an overview of an ongoing OMG standard specifying precise semantics for UML composite structures and their extensions (e.g., profiles). The specification includes semantic definitions for all the metaclasses supporting the ability of classifiers to have both an internal structure (comprising a network of linked parts) and an external structure (consisting of one or more ports). It covers both structural semantics (e.g. the runtime manifestations of connectors, ports, and parts) and behavioral semantics (e.g. life-cycles of composite objects and their constituents, the nature and characteristics of flows through ports and connectors). It builds on the precise semantics of fUML, which specifies execution semantics of a computationally complete and compact subset of UML 2 to support execution of activities.

Visualisation orientée modèle de trace de simulation

El Arbi Aboussoror, Ileana Ober, Iulian Ober

IRIT, Université de Toulouse, 118 Route de Narbonne
F -31062 Toulouse, France
{El-Arbi.Aboussoror, Ileana.Ober, Iulian.Ober}@irit.fr

Résumé Plusieurs techniques et outils de vérification de modèle existent et pourtant leur utilisation dans l'industrie reste très limitée. De nombreuses raisons peuvent expliquer ce manque d'engouement pour de telles techniques, mais une des raisons principales est l'exploitation, aujourd'hui difficile, des résultats du processus de vérification.

Dans cette présentation je parlerai d'une approche d'aide au diagnostic d'erreur qui améliore l'exploitation des résultats de vérification en introduisant des techniques de visualisation. L'approche a été implémentée et intégrée á l'outil de validation de modèle UML IFx-OMEGA [1]. Ces travaux s'incrivent dans un effort de recherche plus large pour rendre les approches de vérification de modèle plus abordables.

1. http://www.irit.fr/ifx/

IDM et côntrole-commande nucléaire : défis et enjeux dans le cluster CONNEXION

Catherine Devic[1], Jean-Christophe Blanchon[2] Jean-Francois Cabadi[3], Francois-Xavier Dormoy[4], Daniele Lanneau[5], and Valerie Zille

[1] EDF R&D, 6 Quai Watier 78400 Chatou, France
[2] Corys Tess, 44 rue des Berges 38024 Grenoble, France
[3] Alstom Power Automation and Controls, 14 rue Jean Bart 91345 Massy, France
[4] Esterel Technologies, France
[5] Atos Worldgrid, France
[6] AREVA, 1 Place Jean Millier 92084 Paris, France

Résumé Le Cluster CONNEXION (CÔntrole Commande Nuclaire Numrique pour lEXport et la renovatION) a pour ambition de proposer et de valider les principes de design d'une architecture innovante de plateformes modulaires de côntrole-commande adapte aux centrales nuclaires en France et á l'International. Cette architecture intégre un ensemble de briques technologiques développées par les partenaires acadmiques (CEA, INRIA, CNRS/CRAN, ENS Cachan, LIG, Telecom ParisTech) et reposant sur des collaborations entre des grands intégrateurs comme AREVA et ALSTOM, l'opérateur EDF en France et des technoproviders de logiciels embarqus (Atos Worldgrid, Rolls-Royce Civil Nuclear, CORYS TESS, Esterel Technologies, All4Tec, Predict). Le cluster CONNEXION s'articule autour d'un objectif de R&D ambitieux intégrant plusieurs innovations majeures pour faciliter la conception modulaire et la rénovation des systémes de contrôle-commande des centrales nucléaires. Les solutions apportées par l'IDM depuis de nombreuses années sont mises ensemble afin de répondre à des besoins identifiés. Par le passé, cette industrie a pu bénéficiér largement des progrés majeurs de l'ingénierie dirigée par les modéles à la fois pour concevoir les premiers systémes de protection numériques, justifier de leur confiance tout au long de leur cycle de développement mais aussi pour arriver intégrer la premiére salle de commande informatisée. Pour demain, l'ambition est d'être en mesure de disposer d'architectures et de solutions modulaires capables de s'adapter aux rglementations locales de chaque pays, de faciliter la retro-ingénierie de systémes de côntrole-commande que l'on n'a pas construit, définir des méthodes et ateliers de développement, de validation et de (re-)qualification de ces systémes tout en conservant la maîtrise des données tout au long du cycle de vie des installations. L'expérience dégagée dans le cluster CONNEXION permet donc d'offrir un état des lieux des pratiques de l'IDM et de proposer des axes de recherches innovants essentiels pour les années à venir pour parfaire sa diffusion dans les pratiques industrielles.

Session du groupe de travail LaMHA

Langages et Modèles de Haut-niveau pour la programmation parallèle, distribuée, de grilles de calcul et Applications

Broadcast Algorithms for CAN: Design and Mechanisation

Francesco Bongiovanni[1] and Ludovic Henrio[2]

[1] Joseph Fourier University and LIG Labs, Grenoble
[2] INRIA-I3S-CNRS, University of Nice Sophia Antipolis

Context and Objectives

A *CAN* [4] (Content-Addressable Network) is a structured P2P network based on a d-dimensional Cartesian coordinate space labelled $\mathcal{D}$. This space is dynamically partitioned among all peers in the system such that each node is responsible for storing data, in the form of *(key, value)* pairs, in a sub-zone of $\mathcal{D}$. To store a (k, v) pair, the key k is deterministically mapped onto a point in $\mathcal{D}$ and the value v is stored by the node responsible for the zone comprising this point. The search for the value corresponding to a key k is achieved by applying the same deterministic function on k to find the node responsible for storing the corresponding value. These two mechanisms are performed by an iterative routing process starting at the query originator and always going from a zone to a neighbouring zone.

Our aim is to design an *efficient* (in terms of number of messages) and *correct* broadcast algorithm for the CAN overlay network. We use mechanical proofs to ensure the correctness of the studied protocols, with a much higher confidence than paper proofs. We provide an Isabelle/HOL framework to study CAN and broadcast algorithms on those networks.

To our knowledge the main proposals for efficient broadcast in a CAN network, and the closest works to our, are M-CAN [5] and Meghdoot [2]. M-CAN [5] is an application-level multicast primitive which is almost efficient, but it does not eliminate all duplicates if the space is not perfectly partitioned and the dimension is greater than two. The authors measured 3% of duplicates on a realistic example. In a publish/subscribe context, Meghdoot [2], built atop CAN, also proposes a mechanism that totally avoids duplicates but require the dissemination to originate from one corner of the zone to be covered. Compared to those approach, our algorithm can originate from any node of the CAN and still remove all the duplicates.

To our knowledge, we are the first ones to formalise and prove some properties of an abstraction of the CAN overlay network using a theorem prover. Our mechanised model should greatly increase the correctness and understanding of distributed algorithms for structured P2P networks, and the confidence one has in their correctness.

Overview of the Mechanisation

A first crucial question when formalizing a complex structure like a CAN is which level of abstraction should be used, and which notions of Isabelle/HOL should represent basic notions of CAN networks. We chose to represent a CAN by a set of nodes, a zone for each node, and a neighboring relationship, stating whether any two nodes are neighbors. More precisely, a *CAN* is a set of integers identifying the different nodes. A function *CZ* matches each node to a *Zone*; a *zone* is simply a set of points, where each point is represented by a tuple of integers: *CZ N* is the zone under the responsibility of the node *N*. Also we require that the set of nodes is finite and the set of their zones partitions the whole space into disjoint zones covering the whole space. Each node is responsible for a zone that never changes and is called the zone of a node; note that our broadcast algorithm will also rely on some zones, i.e. sets of points.

The structured network represented is more general than a CAN: in a CAN, zones are necessarily hyperrectangles, whereas ours could be any tuple set. We prefer relying on a less restrictive definition of the structure to see which properties of our algorithm are verified in those conditions and also to lay down the groundwork for future challenges such as node churn. Later, requirements on the structure can be added to prove further properties, e.g. an algorithm may only be efficient if

the zones are hyperrectangles. Difficult parts of the formalization concern reasoning by induction on a set that is finite but not inductively defined. To ease this kind of reasoning when dealing with zones, we developed an induction principle based on the number of nodes inside a zone. In our framework, we reason on the topology of the network, defined by a neighbor relation, and related lemmas deal with the notion of *connected* zone, i.e. a node where any two nodes can (indirectly) communicate. We used this framework to describe a class of broadcast algorithms that relies on the notion of "zone to be covered".

Defining a broadcast in a natural way using Isabelle/HOL is not trivial; here we decide to put an emphasis on the way messages are processed. Our formalization is centered around the specification of messages which are the *consequences* of a given message and on the specification of the *set of messages* used to broadcast the original message. Then we define the way messages are broadcasted by an inductive definition, where messages are "processed" one message after the other sequentially. In our formalization,

The main properties we aim at are *coverage* and *optimality* (in term of number of messages) for some specific broadcast algorithm. We focus on broadcast algorithms that rely on zones to be covered by the consequence of a message. The idea is that each message is given a zone and the messages that are triggered by this message must cover this zone, but should not pass by nodes outside this zone. Finding a partition of a zone that is both connected (i.e., there are communication paths between any two nodes of the zone), and ensures efficiency is the tricky part here. We proved that such a partition can be found; indeed, a simple but efficient algorithm can be designed as follows: suppose a node N receives a message with a given zone Z to cover; we split this zone into several zones Z_i, where each zone is connected, no zone touches another, and each zone contains a node N_i neighbour of N. We first prove that such a decomposition necessarily exists. This partition ensures that any node can receive the message from a single source and ensures efficiency in term of number of messages: each node receives the message a single time. Even if the partition we exhibit results in a broadcast algorithm that is not efficient in term of latency, we proved formally that an efficient algorithm exists. In the meantime we also designed an algorithm that is both optimal in term of number of messages and features a reasonable latency [3].

The current specification and proofs consist of almost 5000 lines of Isabelle/HOL[3], for more than 150 lemmas and theorems. The length of the proofs is however not uniform: simple properties on the network or the connectivity could take a couple of lines, whether advanced properties on connectivity, and most of the properties of the broadcast algorithm require dealing with a lot of cases, or rely on complex inductions, they necessitate several hundreds of lines.

Overall, *we proved that there exists an algorithm that covers the whole CAN network without sending twice a message to the same node* [1]. This development also shows the capabilities of our Isabelle/HOL framework. We also designed an optimal algorithm that features a better latency than the naive one we exhibited for the proof [3].

References

1. Francesco Bongiovanni and Ludovic Henrio. A mechanized model for can protocols. In *16th International Conference on Fundamentals of Software Engineering (FASE'13)*, LNCS. Springer, 2013. To appear.
2. A. Gupta, O.D. Sahin, D. Agrawal, and A.E. Abbadi. Meghdoot: content-based publish/subscribe over P2P networks. In *Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware*, pages 254–273. Springer-Verlag New York, Inc., 2004.
3. Ludovic Henrio. *Formal Models for Programming and Composing Correct Distributed Systems*. PhD thesis, Université de Nice Sophia-Antipolis, July 2012. HDR Thesis.
4. S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content-Addressable Network. In *SIGCOMM*, pages 161–172. ACM, 2001.
5. S. Ratnasamy, M. Handley, R. Karp, and S. Shenker. Application-level multicast using content-addressable networks. *Networked Group Communication*, 2001.

[3] see: www-sop.inria.fr/oasis/personnel/Ludovic.Henrio/misc.

Boost.SIMD: Generic Programming for Portable SIMDization

Pierre Estérie
LRI, Université Paris-Sud XI
Orsay, France
pierre.esterie@lri.fr

Mathias Gaunard
Metascale
Orsay, France
mathias.gaunard@metascale.org

Joel Falcou
LRI, Université Paris-Sud XI
Orsay, France
joel.falcou@lri.fr

Jean-Thierry Lapresté
IP (Institut Pascal), Université Blaise Pascal
Clermont-Ferrand, France
lapreste@univ-bpclermont.fr

Brigitte Rozoy
LRI, Université Paris-Sud XI
Orsay, France
rozoy@lri.fr

ABSTRACT

SIMD extensions have been a feature of choice for processor manufacturers for a couple of decades. Designed to exploit data parallelism in applications at the instruction level and provide significant accelerations, these extensions still require a high level of expertise or the use of potentially fragile compiler support or vendor-specific libraries. In this poster, we present Boost.SIMD, a C++ template library that simplifies the exploitation of SIMD hardware within a standard C++ programming model.

Categories and Subject Descriptors

C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures (Multiprocessors)—*Single-instruction-stream, multiple-data-stream processors (SIMD)*; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Software libraries*

Keywords

SIMD, C++, Generic Programming, Template Metaprogramming

1. MOTIVATIONS

Since the late 90's, processor manufacturers have provided specialized processing units called multimedia extensions or Single Instruction Multiple Data (SIMD) extensions. The introduction of this feature has allowed processors to exploit the latent data parallelism available in applications by executing a given instruction simultaneously on multiple data stored in a single special register. Today's processor architectures offer rich SIMD instruction sets working with larger and larger SIMD registers. For example, the AVX extension introduced in 2011 enhances the x86 instruction set for the Intel Sandy Bridge and AMD Bulldozer microarchitectures by providing a distinct set of 16 256-bit registers. Similary, the forthcoming Intel MIC Architecture will embed 512-bit SIMD registers and embedded systems also integrate such features like NEON ARM extensions. However, programming applications that take advantage of the SIMD extension remains a complex task. Programmers that use low-level intrinsics have to deal with a verbose programming style which is burying the initial algorithms in architecture specific implementation details. Furthermore, these efforts have to be repeated for every different extension that one may want to support, making design and maintenance of such applications very time consuming. Different approaches have been suggested to limit these shortcomings. On the compiler side, **autovectorizers** implement code analysis and transform phases to generate vectorized code [2, 4]. Compilers are able to detect code fragments that can be vectorized but they struggle when the classical code is not presenting a clear vectorizable pattern (complex data dependencies, non-contiguous memory accesses, aliasing or control flows). Other compiler-based systems use code directives to guide the vectorization process by enforcing loop vectorization. The ICC and GCC extension `#pragma simd` is such a system. To use this mechanism, developers explicitly introduce directives in their code, thus having a fine grain control on where to apply SIMDization. However, the generated code quality will greatly depend on the used compiler. Another approach is to use **libraries** like Intel MKL. Those libraries offer a set of domain-specific routines that are optimized for a given architecture. This solution suffers from a lack of flexibility as the proposed routines are optimized for specific use-cases that may not fulfill arbitrary code constraints.

PACT'12, September 19–23, 2012, Minneapolis, MN, USA.
ACM 978-1-4503-1182-3/12/09.

2. THE BOOST.SIMD LIBRARY

The main issue of SIMD programming is the lack of proper abstractions over the usage of SIMD registers. This abstraction should not only provide a portable way to use hardware-specific registers but also enable the use of common programming idioms when designing SIMD-aware algorithms.

2.1 SIMD register abstraction

The first level of abstraction introduced by Boost.SIMD is the `pack` class. For a given type `T` and a given static integral value `N` (`N` being a power of 2), a `pack` encapsulates the best type able to store a sequence of `N` elements of type `T`.

For arbitrary `T` and `N`, this type is simply `std::array<T,N>` but when `T` and `N` matches the type and width of a SIMD register, the architecture-specific type used to represent this register is used instead. By default, `pack` will automatically select a value that will trigger the selection of the native SIMD register type. The `pack` class handles these low-level SIMD register type as regular objects with value semantics, which includes the ability to be constructed or copied from a single scalar value, list of scalar values, iterator or range. In each case, the proper register loading strategy (splat, set, load or gather) will be issued. This abstraction is coupled with a large set of functions covering the classical set of operators along with a sensible amount (200+) of mathematical functions and utility functions like constant generators and arithmetic/IEEE 754/reductions functions. A fundamental aspect of SIMD programming relies on the effective use of fused operations like multiply-add on VMX extensions or sum of absolute differences on SSE extensions. `pack` relies on *Expression Templates* [1] to capture the Abstract Syntax Tree (AST) of large `pack`-based expressions and performs compile-time optimization on this AST. These optimizations include the detection of fused operation and replacement or reordering of reductions versus elementwise operations. Moreover, the AST-based evaluation process is able to merge multiple function calls into a single inlined one, contrary to solutions like MKL where each function can only be applied on the whole data range at a time. This increases data locality and ensures high performance for any combination of functions.

2.2 C++ Standard integration

Realistic applications usually require functions applied over a large set of data. To support such a use case, the library provides a set of classes to integrate SIMD computation inside C++ code relying on the C++ Standard Template Library (STL) components, thus reusing its generic aspect to the fullest. Based on Generic Programming [3], the STL is based on the separation between data, stored in various `Containers`, and the way one can traverse these data, thanks to `Iterators` and algorithms. Boost.SIMD reuses existing STL Concepts to adapt STL-based code to SIMD computations by providing SIMD-aware allocators, iterators for regular SIMD computations and hardware-optimized algorithms. The hardware implementation of SIMD processing units introduces constraints related to memory handling. Performance is guaranteed by accessing the memory through dedicated `load` and `store` intrinsics that perform register-length aligned memory accesses. This constraint requires a special memory allocation strategy via OS and compiler-specific function calls. Boost.SIMD provides a STL compliant allocator dealing with this kind of alignment. The `simd::allocator` class wraps these OS and compiler functions in a simple STL-compliant allocator. Usually, modern C++ programming style based on Generic Programming leads to an intensive use of various STL components like Iterators. Boost.SIMD provides iterator adaptors that turn regular random access iterators into iterators suitable for SIMD processing. These adaptors act as free functions taking regular iterators as parameters and return iterators that output `pack` whenever dereferenced. These iterators are then usable directly with common STL algorithms such as `transform` or `fold`. The code keeps a conventional structure which facilitates the usage of template functors for both scalar and SIMD cases, maximizing code reuse.

3. THE RGB2YUV ALGORITHM

The RGB and YUV models are both color spaces with three components that encode images or videos. Unlike RGB, the YUV color space takes into account the human perception of the colors. The RGB2YUV algorithm fits the data parallelism requirement for SIMD computation. The comparison shown in Table 1 shows the performance of Boost.SIMD against scalar C++ code and SIMD reference code. The implementation is realized in single precision floating-point and the results are presented in cycles per point (cpp).

Table 1: Results for RGB2YUV algorithm in *cpp*

Size	Version	SSE4.2	AVX	Altivec
256^2	Scalar C++	29.23	21.46	42.51
	Ref. SIMD	6.48	2.80	29.05
	Boost.SIMD	6.51	2.45	29.01
	Speedup	4.49	8.76	1.47
	Overhead	0.04%	-14.3%	0.1%

The speedups obtained with SEE4.2 and AVX are superior to the expected $\times 4$ and $\times 8$ on such extensions and no overhead is added by Boost.SIMD. A slowdown appears with Altivec due to the lack of a level 3 cache which causes the SIMD unit to wait constantly for data from the main memory. For a size of 64^2 on the PowerPC, Boost.SIMD performs at 4.65 cpp against 36.32 cpp for the scalar version giving a speedup of 7.81 which confirms the memory limitation of such an architecture. The latent data parallelism in the algorithm is fully exploited and the benchmarks corroborate the ability of the library to generate effecient SIMD code.

4. CONCLUSION

SIMD instruction sets are a technology present in an ever growing number of architectures. Despite the performance boost that such extensions usually provide, SIMD has been usually underused. Losing the $\times 4$ to $\times 16$ speed-ups they may provide in HPC applications is starting to be glaring. We presented Boost.SIMD, which aims at simplifying the design of SIMD-aware applications while providing a portable high-level API, integrated with the C++ Standard Template Library and solves the problem of portable SIMD code generation.

5. REFERENCES

[1] K. Czarnecki, U. W. Eisenecker, R. Glück, D. Vandevoorde, and T. L. Veldhuizen. Generative programming and active libraries. In *Generic Programming*, pages 25–39, 1998.

[2] J. Shin. Introducing control flow into vectorized code. *Parallel Architectures and Compilation Techniques, International Conference on*, 0:280–291, 2007.

[3] A. Stepanov and M. Lee. The standard template library. Technical Report 95-11(R.1), HP Laboratories, 1995.

[4] H. Wang, H. Andrade, B. Gedik, and K.-L. Wu. A code generation approach for auto-vectorization in the spade compiler. In *LCPC'09*, pages 383–390, 2009.

OSL: an algorithmic skeleton library with exceptions

Joeffrey Legaux

LIFO, Université d'Orléans, France,
Joeffrey.Legaux@univ-orleans.fr

Keywords: Parallel programming, algorithmic skeletons, C++, exceptions

1 Context

Due to the increasing complexity of computer programs and systems, abnormal or exceptional events are very likely to happen. Even in the case of a certified system, input/output operations, memory allocation or in general hardware related operations may cause errors. It is preferable that such events do not cause a crash of the programs or systems. Therefore modern programming languages offer mechanisms to manage such events.

Work on managing exceptions started in the 70's [3]. Nowadays, exception mechanisms are more than a way to manage errors: There are structural parts of the languages and the execution models. Three distinct roles are played by an exception mechanism:

- The capability to continue the execution of a program in the absence of a result that failed to be computed. It may simply be the capability to output the nature of the problem before terminating the execution. One possible solution for this is to output a special return value and to have a different value for each possible error scenario. This is what is done in C for example with the integer value return by the `main` function or the return value of some MPI functions that may indicate errors. However these cases do not belong to the exception mechanism class as they make mandatory the *explicit* treatment of exceptional cases, for all the calls: this pollutes the code and makes it harder to read, understand and maintain. An exception mechanism introduces an *implicit* treatment of exceptional cases. Code is kept smaller and dealing with exceptional cases using specific language constructions makes the code clearer.
- When an exception is raised, the normal execution flow is interrupted. The execution flow is routed to a specific code to deal with the exceptional case at hand. Therefore an exception should trigger a specific treatment without step-by-step going through the call stack as in normal function calls.
- A catch or retry mechanism allows to return to a safe program state or to try again the operation with different parameters.

To have an idea of the complexity of exception handling in the case of a parallel exception, it may be helpful to think of exception propagation as a jump in the call stack of a program, or successive jumps to an adequate routine to handle the exception. In the parallel case there exists one stack by processor or process and there stacks may be entirely independent if we assume no underlying structure. In the absence of synchronisation mechanisms, the notion of the execution state at a given time is itself difficult to define as every machine has its own clock.There are a lot of work on exception in a concurrent context, there is currently no widely accepted best solution [5]. The exception mechanism should be adapted to the considered concurrent or parallel model. In most systems, in particular concurrent or parallel ones, exception handling is done locally or sequentially, and cannot guarantee the global coherence of the system after an exception is caught. In this context, we have the advantage to work with a *structured parallel model*: algorithmic skeletons [1].

2 Our domain : algorithmic skeletons

Skeletal parallel languages or libraries provide a finite set of *algorithmic skeletons* that are higher-order functions or patterns that can be executed in parallel. A skeleton often captures the pattern

of a classical parallel algorithm such as a pipeline, a parallel reduction, or parallel operations on distributed collections. In such a context data-structures are considered *globally for the whole parallel machine*, even in the case of distributed memory machine. This is the *global view* also advocated by the Chapel language designers [2]. It eases parallel programming compared to the fragmented view provided by SPMD programming (for example MPI). Usually the sequential semantics of the skeleton is simple and corresponds to the usual semantics of similar higher-order functions in functional programming languages or usual operations and iterations on collections in object oriented languages. The user of algorithmic skeletons has just to compose some of the skeletons to write her parallel application. Orléans Skeleton Library or OSL is an efficient algorithmic skeleton library for C++ that uses meta-programming techniques to attain good performances when composing skeletons [4].

3 Exceptions in OSL

I will first present the design of an exception mechanism adapted to the parallel execution model of Orléans Skeleton Library that ensures the global coherence of the system after exceptions are caught. This mechanisms allows the user of our library to encapsulate the parallel skeleton calls in exception catching blocks the same way he would have designed a catching block around calls to sequential functions.

I will then present some details of the implementation of this mechanism in C++. The way exceptions are handled in this language raised several issues : to transmit them between computing nodes we need to be able to serialize them, a functionality which is not provided by the standard library. Furthermore, we have to be able to take into account their polymorphic nature through this serialization. The Boost framework in particular helped us in achieving this goal. Experimental results show that this mechanism induces a moderate performance penalty. Although this varies with the number of computing nodes and the length of the considered computation, it seems to peak at around 50% on a large number of nodes with a sufficiently large computation.

I will finally illustrate the additionnal expressiveness provided by this mechanism with an implementation of a parallel backtracking algorithm.

Acknowledgements This is a joint work with Frederic Loulergue and Sylvain Jubertie (LIFO). This work is partly supported by ANR (France) and JST (Japan) (project PaPDAS ANR-2010-INTB-0205-02). Joeffrey Legaux is supported by a PhD grant from the *Conseil Général du Loiret*.

References

1. Cole, M.: Algorithmic Skeletons: Structured Management of Parallel Computation. MIT Press (1989), available at `http://homepages.inf.ed.ac.uk/mic/Pubs`
2. Deitz, S.J., Callahan, D., Chamberlain, B.L., Snyder, L.: Global-view abstractions for user-defined reductions and scans. In: PPoPP. pp. 40–47. ACM, New York, NY, USA (2006)
3. Goodenough, J.B.: Exception handling: issues and a proposed notation. Communications of the ACM 18(12), 683–696 (1975)
4. Javed, N., Loulergue, F.: Parallel Programming and Performance Predictability with Orléans Skeleton Library. In: International Conference on High Performance Computing and Simulation (HPCS). pp. 257–263. IEEE (2011)
5. Romanovsky, A., Kienzle, J.: Action-oriented exception handling in cooperative and competitive concurrent object-oriented systems. In: Romanovsky, A.B., Dony, C., Knudsen, J., Tripathi, A. (eds.) Advances in Exception Handling Techniques, LNCS, vol. 2022, pp. 147–164. Springer, New York, NY, USA (2001)

Analyse statique de programmes concurrents et dynamiques
Une approche par le π-calcul

Aurélien Deharbe et Frédéric Peschanski

Université Pierre et Marie Curie - LIP6 - APR
prenom.nom@lip6.fr

1 Introduction

Dans ce travail, nous nous intéressons à l'analyse statique de propriétés comportementales d'un langage parallèle de haut-niveau, permettant notamment d'exprimer la récursion, les fonctions d'ordre supérieur ou encore la création dynamique de *threads*. Les propriétés qui nous intéressent sont du type : ce programme mène-t-il à un interblocage ? A-t-il des fuites mémoire ? Se terminera-t-il ? Nous attachons également une importance à la nature compositionnelle de nos analyses, qui permet l'étude de fragments de programmes, et facilite ainsi le passage à l'échelle.

Le π-calcul est une algèbre de processus [2], qui permet d'exprimer intrinsèquement le parallèlisme et la concurrence. En autorisant l'échange de noms de canaux de communication eux-mêmes, il offre une grande expressivité permettant alors de modéliser, entre autres phénomènes dynamiques, ceux du langage de haut-niveau que l'on souhaite analyser : récursion, fonctions de premier ordre, création dynamique de ressources. Etant également compositionnel, il s'impose ici comme un langage de choix pour la modélisation.

Le π-calcul étant un langage très expressif, les techniques de vérification qui le ciblent sont souvent *ad hoc* et complexes [5]. La difficulté principale concerne la génération d'un espace d'états (système de transitions étiquetées, ou LTS) analysable statiquement. Contrairement aux *HD-automata* [3], notre approche ne requiert pas de réviser le cadre théorique global. Notre principale contribution est une brique algorithmique, le ramasse-miettes omniscient (OGC), qui permet de calculer un espace d'états "optimal" - un automate *ground* - pour la vérification de propriétés portant sur les ressources dynamiques d'un fragment de programme.

2 Un automate *ground* pour le π-calcul

Plutôt que de développer des algorithmes spécifiques à la vérification de propriétés sur des modèles issus de processus en π-calcul, notre objectif ici est de produire des modèles génériques, sur lesquels appliquer par la suite les algorithmes "classiques", et plus efficaces : par exemple du *model checking* ou bien de la vérification de bisimulation [4].

Plusieurs problèmes se posent alors pour la construction de l'automate *ground*. Un premier obstacle intervient pour les transitions résultantes d'une réception : le domaine des noms est infini, mais il faut limiter le nombre de noms que le processus considéré peut recevoir afin de conserver un nombre fini d'états. Une solution est de ne considérer que les noms qui seront effectivement utilisés par la suite du processus (les noms actifs), ainsi qu'un nom "frais", représentant les noms inconnus du processus, issus de l'environnement extérieur. Il a déjà été prouvé que ces instanciations lors d'une réception sont suffisantes [1], la difficulté étant maintenant de déterminer l'ensemble des noms actifs, cet ensemble n'étant pas toujours équivalent à l'ensemble des noms libres dans le terme syntaxique du processus correspondant.

Un second problème intervient lors de la création dynamique de nom, et également lors du choix d'un nom frais pour une réception évoqué juste avant. Il s'agit cette fois de garantir la fraîcheur d'un nom [6], tout en conservant la propriété selon laquelle deux processus bisimilaires opéreront le même choix. Après avoir fixé un ordre sur les noms, on choisira d'élire un nouveau nom en réutilisant le plus petit nom qui n'est plus actif par la suite. Cette technique s'apparente à celle d'un ramasse-miettes puisque l'on récupère les noms qui ne sont plus utilisés, mais avec une récupération optimale qui prévoierait à l'avance si une ressource peut être réutilisée ou non.

3 Le ramasse-miettes omniscient (l'OGC)

En réponse à la problématique exposée dans le paragraphe précédent, nous proposons le *framework* du ramasse-miettes omniscient (l'OGC). Il est constitué d'un algorithme en plusieurs passes, qui permet d'attribuer les **meilleures instanciations** possibles à des **ressources dynamiques**. Ce *framework* constitue en fait un cadre algorithmique intermédiaire entre le LTS et l'automate *ground*, la contrainte étant de satisfaire dans tous les cas le théorème suivant :

Théorème 1 *Deux processus en π-calcul sont bisimilaires (bisimulation du π-calcul) si et seulement si leurs automates respectifs générés au travers de l'OGC sont bisimilaires (bisimulation des automates ground)*

Afin de ne travailler que sur un sous-ensemble des états du LTS à normaliser, nous abstrayons les ressources dynamiques dans des structures de données adaptées. Le premier modèle manipulé consiste en un graphe enraciné encodant une relation causale sur trois types d'événements sur les ressources dynamiques : allocations, utilisations et libérations. Après une détection des composantes fortement connexes, nous sommes en mesure de supprimer les cycles et de compléter le graphe afin de former un treillis complet, sur lequel nous définissons alors de manière inductive les ensembles de ressources effectivement actives à un instant donné, et les ensembles de conflits entre ressource (ressources nécessitants obilgatoirement deux identités distinctes). Ces informations pourront être collectées par un algorithme linéaire de parcours du treillis. Des conflits et de l'ordre partiel induit par l'ordre d'apparition des ressources se forment enfin des classes d'équivalence regroupant les ressources pouvant partager les mêmes instanciations. Nous utilisons finalement un espace de nom séparé, basé sur une horloge logique, pour attribuer les meilleures instanciations possibles aux ressources dynamiques, et répercuter ces instanciations des ressources dans le LTS.

Le *framework* présenté dépasse le cadre du π-calcul, par la généralité des structures de données qui la composent. Il s'applique à tout formalisme traitant, de manière abstraite ou non, de ressources dynamiques. Dans le contexte initialement proposé de l'analyse de programmes dynamiques, on s'intéressera par la suite à l'étude de détection de terminaison et de divergences, et plus généralement des propriétés relatives à l'utilisation des ressources.

Références

1. Huimin Lin. Computing bisimulations for finite-control pi-calculus. *J. Comput. Sci. Technol.*, 15(1):1–9, 2000.
2. Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, i and ii. *Inf. Comput.*, 100(1):1–40, 1992.
3. Ugo Montanari and Marco Pistore. Structured coalgebras and minimal hd-automata for the *pi*-calculus. *Theor. Comput. Sci.*, 340(3):539–576, 2005.
4. Robert Paige and Robert Endre Tarjan. Three partition refinement algorithms. *SIAM J. Comput.*, 16(6):973–989, 1987.
5. Marco Pistore and Davide Sangiorgi. A partition refinement algorithm for the pi-calculus. *Inf. Comput.*, 164(2):264–321, 2001.
6. Fernando Rosa-Velardo and David de Frutos-Escrig. Decidability results for restricted models of petri nets with name creation and replication. In *Petri Nets*, pages 63–82, 2009.

Session du groupe de travail LTP

Langages, Types et Preuves

Vérification de systèmes paramétrés avec Cubicle

Sylvain Conchon[1,2], Alain Mebsout[1,2] et Fatiha Zaïdi[1]

[1] LRI, Université Paris Sud, CNRS, Orsay F-91405
[2] INRIA Saclay – Île-de-France F-91893 Orsay

Résumé Cubicle est un model-checker pour vérifier des propriétés de sûreté d'algorithmes faisant intervenir un nombre quelconque de processus. Ces algorithmes sont décrits sous forme de systèmes de transitions dits *paramétrés*. Les propriétés de sûreté ainsi que les états du système sont décrits par des formules logiques du premier ordre. Pour vérifier ces propriétés, Cubicle met en œuvre un algorithme d'atteignabilité par chaînage arrière qui utilise un démonstrateur SMT (Satisfiabilité Modulo Théories). Les expériences, menées sur la vérification d'algorithmes d'exclusion mutuelle et de protocoles de cohérence de cache, montrent que Cubicle est efficace et compétitif avec les *model checkers* de la même famille. Cubicle est un logiciel libre développé en OCaml. Il utilise le démonstrateur Alt-Ergo zero, une version allégée et améliorée d'Alt-Ergo. Une implantation parallèle se reposant sur la bibliothèque Functory est également proposée.

1 Introduction

Le *model checking* consiste à s'assurer qu'un modèle, représentant un programme ou l'abstraction d'un système complexe, vérifie certaines propriétés. Les outils de vérification par *model checking* sont nombreux et diffèrent selon la nature des modèles à analyser, les techniques pour les représenter et enfin les propriétés à vérifier. Ainsi, on distingue par exemple les *model checkers* qui analysent des programmes dans un langage comme C (Blast [21]) ou Java (JavaPathFinder [20]) de ceux qui manipulent des automates (Spin [22]) ou des systèmes de transition avec un nombre d'états fini (Murφ [14]) ou infini (TReX [4]). La représentation des états peut être énumérative (les états sont alors représentés de manière individuelle) ou symbolique (dans ce cas, on manipule des ensembles d'états, par exemple à l'aide de BDD comme dans NuSMV [10]). On distingue également le type des propriétés à vérifier (sûreté, vivacité ou équité) ainsi que les logiques pour les exprimer (LTL, CTL, CTL* etc.).

Cubicle [13] appartient à la famille des *model checkers* symboliques qui vérifient des propriétés de sûreté sur des systèmes de transition infinis et faisant intervenir un nombre quelconque de processus. Ces systèmes sont dits *paramétrés*. Par exemple Cubicle est capable de prouver des propriétés d'exclusion mutuelle d'un algorithme à n threads pour tout n. Un des avantages de cette approche

paramétrée est qu'il est souvent bien plus facile de prouver une propriété pour un nombre arbitraire de processus que de prouver cette propriété pour un nombre fixé lorsque celui-ci devient grand. Malheureusement, vérifier la sûreté d'un système paramétré est en général indécidable [5]. Aussi, il est important d'identifier un fragment sur lequel il est possible de prouver les propriétés qui nous intéressent. La classe des systèmes traitée par Cubicle est celle des *systèmes à tableaux* [16] dont la sûreté est décidable sous certaines conditions [17]. C'est une classe syntaxiquement restreinte de systèmes de transition paramétrés dont les états sont représentés par un ensemble fini de tableaux indexés par un nombre arbitraire d'identificateurs de processus. Les protocoles de cohérence de cache et les algorithmes d'exclusion mutuelle sont des exemples typiques de systèmes dont les états peuvent être représentés avec des tableaux. Cubicle vérifie la sûreté de tels programmes de la manière suivante : les états initiaux ainsi que les états *dangereux* du système sont représentés par des formules logiques et l'analyse consiste à vérifier qu'aucun état *dangereux* n'est atteignable à partir d'un état initial en appliquant les instances des transitions du système. Pour cela, cet algorithme calcule une clôture de la pré-image de la formule *dangereuse* par la relation de transition à l'aide d'un démonstrateur SMT. Ce calcul est rendu possible car on manipule des formules spécifiques (conjonctions de littéraux existentiellement quantifiées) représentant un ensemble infini d'états qu'on appelle *cubes*. La sûreté est garantie si aucun de ces cubes ne contient un des états initiaux.

L'algorithme d'atteignabilité de Cubicle est directement issu du cadre théorique proposé par Ghilardi et Ranise [17]. Cependant, il faut aller au delà de ces travaux si on souhaite prouver la sûreté de systèmes paramétrés donnés comme défis à la communauté du model checking. Par exemple, la vérification du protocole de cohérence de cache proposé par Steve German, et décrit dans [7], nécessite l'implantation de nombreuses optimisations. Dans cet article, on présente les détails et les optimisations importantes qui permettent à Cubicle de prouver la sûreté d'un tel protocole.

La famille des model checkers pour systèmes paramétrés n'est pas très fournie mais elle compte parmi ses rangs des outils qui travaillent sur différents fragments décidables, tels que MCMT [18], Undip [3] ou PFS [2]. Le plus proche concurrent de Cubicle est MCMT (fondé sur les mêmes travaux théoriques). Tout en étant aussi compétitif, Cubicle propose une implantation libre, un langage d'entrée simple d'usage ainsi qu'une architecture parallèle.

Cet article étend la présentation de Cubicle faite dans [13] en donnant plus de détails sur l'implémentation et les propriétés sur lesquelles se basent nos optimisations (en particulier, les critères pour garantir la terminaison de l'algorithme d'atteignabilité). Nous présentons également l'interface OCaml d'Alt-Ergo ZERO, le démonstrateur automatique SMT utilisé dans Cubicle.

Cet article présente le langage d'entrée de Cubicle en section 2 et les détails d'implantation en section 3. Son architecture parallèle, décrite en section 4, repose sur la bibliothèque Functory [15]. Cubicle utilise le démonstrateur SMT Alt-Ergo ZERO, une version allégée et améliorée d'Alt-Ergo [11] qui se présente

sous la forme d'une bibliothèque OCaml autonome décrite en section 5. Cubicle est également écrit en OCaml et il est disponible sous licence Apache à l'adresse `http://cubicle.lri.fr`.

2 Langage d'entrée

Le langage d'entrée de Cubicle est une version typée du langage de Murφ [14] et similaire à Uclid [8]. Bien que limité pour l'instant, il est assez expressif pour permettre de décrire aisément des systèmes paramétrés conséquents (75 transitions, 40 variables et tableaux pour le protocole FLASH [23] par exemple).

La description d'un système dans Cubicle commence par des déclarations de types, de variables globales et de tableaux. Cubicle connaît quatre types en interne : le type des entiers (`int`), le type des réels (`reals`), le type des booléens (`bool`) et le type des identificateurs de processus (`proc`). Les tableaux ont la contrainte qu'ils doivent être indexés par des variables de type `proc`. L'utilisateur a aussi la liberté de déclarer ses propres types abstraits ou ses propres types énumérés. L'exemple suivant définit un type énuméré `state` à trois constructeurs `Idle`, `Want` et `Crit` ainsi qu'un type abstrait `data`.

```
type state = Idle | Want | Crit
type data
```

Dans ce qui suit on déclare une variable globale `Timer` de type `real` et trois tableaux indexés par le type `proc`.

```
var Timer : real
array State[proc] : state
array Chan[proc] : data
array Flag[proc] : bool
```

Les états initiaux du système sont définis par une conjonction de littéraux, universellement quantifiée. Ces littéraux caractérisent les valeurs de certains tableaux et variables. Dans un souci de simplicité, l'exemple suivant définit les états initiaux comme ceux ayant leur tableau `Flag` à `False` pour tout processus `z`, `State` à `Idle` et leur variable globale `Timer` valant `0.0`.

```
init(z) { Flag[z] = Flase && State[z] = Idle && Timer = 0.0 }
```

Les propriétés de sûreté à vérifier sont exprimées sous forme négative comme des formules caractérisant les états *dangereux*. Chaque formule dangereuse (ou *unsafe*) doit être un *cube*, i.e. être de la forme $\exists \bar{x}.(distinct(\bar{x}) \wedge C)$, où $\bar{x}$ est un

ensemble de variables représentant des identificateurs de processus, $distinct(\bar{x})$ est la conjonction des différences entre les variables de $\bar{x}$ (exprimant que ces variables doivent être deux à deux distinctes) et C est une conjonction de littéraux. Dans la syntaxe concrète, les quantificateurs existentiels sont laissés implicites, ainsi que le $distinct(\bar{x})$. La formule *dangereuse* suivante exprime que les mauvais états du système sont ceux où il existe deux processus distincts `x` et `y` tels que le tableau `State` contienne la valeur `Crit` à ces deux indices.

```
unsafe(x y) { State[x] = Crit && State[y] = Crit }
```

Le reste du système est donné comme un ensemble de transitions de la forme garde/action. Chaque transition peut être paramétrée par un ou plusieurs identificateurs de processus comme dans l'exemple suivant.

```
transition t (i j)
requires { i < j && State[i] = Idle && Flag[i] = Flase &&
           forall_other k.
                   (Flag[k] = Flag[j] || State[k] <> Want) }
{
  Timer := Timer + 1.0;
  Flag[i] := True;
  State[k] := case
              | k = i : Want
              | State[k] = Crit && k < i : Idle
              | _ : State[k];
}
```

Ici, les paramètres `i` et `j` de la transition sont implicitement quantifiés existentiellement et doivent être les identificateurs de processus deux à deux distincts. Les gardes sont des conjonctions de littéraux (équations, différences, inéquations) et de formules universellement quantifiées de la forme $\forall k.C_1 \vee \ldots \vee C_n$ où k est une variable processus universellement quantifiée distincte de tous les paramètres de la transition et $C_1, \ldots, C_n$ sont des conjonctions de littéraux. Chaque action est une mise à jour d'une variable globale ou d'un tableau. La sémantique des transitions veut que ces mises à jour soient réalisées de manière atomique et donc chaque variable qui apparaît à droite d'un signe `:=` dénote la valeur de cette variable avant la transition. Les mises à jour de tableaux sont codées soit comme de simples affectations `Flag[i] := True` soit par des constructions par cas comme `State[k] := case ...` où la variable `k` est implicitement universellement quantifiée. Dans cette construction `case`, chaque condition doit être une conjonction de littéraux et suppose la négation de toutes les conditions précédentes. Le cas par défaut est dénoté par `_` .

Cette relation de transition (décrite par l'ensemble des transitions) définit l'exécution du système comme une boucle infinie qui à chaque itération :

1. choisit de manière non déterministe une instance de transition dont la garde est vraie dans l'état courant du système
2. met à jour les variables et tableaux d'états conformément aux actions de la transition choisie

Un système est *sûr* si aucun des états dangereux ne peut être atteint à partir d'un des états initiaux.

3 Implantation d'une boucle d'atteignabilité symbolique efficace

Cubicle implante une boucle d'atteignabilité par chaînage arrière pour vérifier les propriétés de sûreté des systèmes à tableaux. Étant donné un programme défini par une formule initiale $\mathcal{I}$ (décrivant les états initiaux du programme), un ensemble de transitions $\mathcal{T}$ et une formule $\mathcal{U}$ représentant les états dangereux, l'algorithme est donnée par le pseudo-code suivant :

```
1  init: V ← ∅
2        Q ← ∅
3        push_queue(Q,U)
4  while not_empty(Q) do
5      φ ← pop_queue(Q)
6      if ¬(φ ∧ I ⊢ ⊥) then return(Unsafe)   (* sûreté *)
7      if ¬(φ ⊢ ⋁_{ψ∈V} ψ) then              (* point fixe local*)
8          V ← V ∪ φ
9          push_queue(Q, pre_T(φ))
10 done
11 return(Safe)
```

Figure 1. Algorithme d'atteignabilité arrière

L'algorithme en figure 1 maintient un ensemble V des nœuds *visités* et une file de priorité Q des nœuds *non visités*. Initialement, V est vide et Q contient la formule dangereuse du système. Ensuite, à chaque itération de la boucle, le cube φ ayant la plus grande priorité est sorti de la file Q et sa sûreté est vérifiée en testant sa cohérence avec la formule initiale (ligne 6). Si ce test de sûreté réussit, alors on enchaîne un test de *subsomption* (ou point fixe local) $\neg(\varphi \wedge \mathcal{I} \vdash \bot)$ (ligne 7). Si ce dernier test échoue, le cube φ est ajouté à l'ensemble des nœuds visités V et on calcule la formule $\texttt{pre}_{\mathcal{T}}(\varphi)$ (ligne 9). Cette formule représente l'ensemble des états à partir desquels il est possible d'atteindre un des états représentés par φ en une transition, autrement dit la pré-image de φ par la relation de transition $\mathcal{T}$. Une des propriété des systèmes à tableaux (indexés par des variables processus) est que si φ est un cube alors $\texttt{pre}_{\mathcal{T}}(\varphi)$ est une disjonction (union) de cubes. Ces cubes sont ensuite ajoutés à la file Q et on

répète une nouvelle itération de la boucle. Au contraire, si le test de *subsomption* est concluant alors φ est ignoré car cela signifie que chaque état représenté par φ est aussi représenté par au moins un cube qu'on a déjà visité. L'algorithme termine soit lorsque un test de sûreté échoue ou bien lorsque la file Q devient vide.

Les tests de satisfiabilité sont envoyés à un démonstrateur SMT (Satisfiabilité Modulo Théories). Les tests de sûreté (ligne 6) sont faciles à prouver car il s'agit simplement de formules closes obtenues par skolémisation. Par contre, les tests de subsomption (ligne 7) de la forme $\neg(\varphi \vdash \bigvee_{\psi \in V} \psi)$ sont plus difficiles. En effet, prouver la validité de l'implication $\varphi \vdash \bigvee_{\psi \in V} \psi$ revient à montrer que la formule $\neg(\neg\varphi \vee \bigvee_{\psi \in V} \psi)$, c'est-à-dire $\varphi \wedge \bigwedge_{\psi \in V} \neg\psi$, est insatisfiable. Puisque φ et ψ sont des formules existentiellement quantifiées de la forme $\exists \bar{x}.F$ et $\exists \bar{y}.G_\psi$ (où F et G_ψ représentent des conjonctions de littéraux), cela revient à manipuler une formule contenant une conjonction de clauses universellement quantifiées $F \wedge \bigwedge_{\psi \in V} \forall \bar{y}.\neg G_\psi$. Le cadre théorique des systèmes à tableaux nous garantit qu'il suffit de considérer l'ensemble Σ des substitutions de $\bar{y}$ vers $\bar{x}$. Par exemple, si on suppose que V contient 20 000 nœuds et que $|\bar{x}| = |\bar{y}| = 5$, il faut alors construire une formule $H = F \wedge \bigwedge_{\psi \in V} \bigwedge_{\sigma \in \Sigma} \forall \bar{y}.\neg G_\psi \sigma$ avec 2,4 millions de clauses. Malgré leur efficacité il n'est pas envisageable d'envoyer de telles formules à un démonstrateur SMT.

Pour passer ce test de subsomption, Cubicle essaye de montrer que H est insatisfiable en la construisant et en la vérifiant incrémentalement. Ceci est fait en examinant toutes les paires (ψ, σ) une à une puis en ajoutant les $\neg G_\psi \sigma$ à la formule H jusqu'à ce qu'elle devienne insatisfiable. Pour chaque paire (ψ, σ), on vérifie si le cube $G_\psi \sigma$ a une des propriétés suivantes :

Proposition 1.

1. *si $G_\psi \sigma$ contient un littéral qui contredit directement un littéral de F alors $G_\psi \sigma$ est redondant dans H pour le test qui nous intéresse et peut donc être enlevé de H*
2. *si $G_\psi \sigma$ est un sous ensemble de F alors H est insatisfiable*

Démonstration.

1. Soit $G_\psi \sigma = g \wedge G'$ et $F = f \wedge F'$ tels que $g \wedge f \vdash \bot$. Autrement dit $f \implies \neg g$, et donc $(f \wedge F') \wedge (\neg g \vee \neg G')$ se réduit en $(f \wedge F')$.
2. Soit $G_\psi \sigma$ un sous ensemble de F, alors $F = G_\psi \sigma \wedge F'$ donc $F \wedge \neg G_\psi \sigma$ est trivialement insatisfiable, donc H est insatisfiable.

Avec la première propriété, on vérifie que le cube $G_\psi \sigma$ n'est pas redondant avant même d'appliquer la substitution σ ; s'il y a redondance le cube est ignoré et une nouvelle paire (ψ, σ) est traitée. Si le cube n'est pas redondant, on lui fait passer le test d'inclusion $G_\psi \sigma \subset F$ de la seconde propriété. Pour calculer efficacement ces tests ensemblistes, les cubes sont représentés à l'aide d'une simple structure de tableau OCaml. Si l'inclusion est vérifiée alors H est déclarée comme étant insatisfiable ; sinon on ajoute $\neg G_\psi \sigma$ à H et le démonstrateur SMT vérifie si la nouvelle version (renforcée) de H devient insatisfiable.

L'intégration de Cubicle avec le démonstrateur SMT au niveau de l'interface de programmation est cruciale pour traiter efficacement ces tests de subsomption. En pratique, on utilise un seul contexte du démonstrateur pour chacun de ces tests ; il est seulement enrichi incrémentalement avec les $\neg G_\psi \sigma$ et sa cohérence est vérifiée à chaque itération. Pour garantir une mise en œuvre la plus efficace et la plus complète possible des tests d'inclusion, les cubes sont maintenus sous forme normale, i.e. les variables sont renommées et les simplifications possibles sont réalisées lors de la construction.

La stratégie d'exploration de l'espace de recherche est elle aussi essentielle. Cubicle gagnera bien souvent à explorer le moins de nœuds possible, ce qui suggère de donner la priorité aux cubes les plus *généraux*, i.e. ceux qui représentent les ensembles d'états les plus grands. Après ce constat, il est clair que des stratégies de recherche naïves comme l'exploration en largeur ou l'exploration en profondeur ne sont pas adaptées. Par défaut, Cubicle utilise une forme de recherche en largeur (BFS) combinée avec une heuristique de retardement pour certains cubes. Dans la stratégie par défaut, un cube est retardé s'il introduit de nouvelles variables de processus ou s'il n'apporte pas d'information supplémentaire sur au moins un tableau. Ces stratégies peuvent être changées en passant différents arguments sur la ligne de commande : l'option `-search` permet de choisir une autre stratégie de recherche comme la recherche en profondeur (DFS) ou une de ses variantes et l'option `-postpone` permet de changer ou de désactiver l'heuristique de retardement. Enfin Cubicle est capable de supprimer des parties de l'arbre de recherche déjà visité, donc des cubes de V, lorsque ceux-ci deviennent subsumés par un nœud plus récent (figure 2).

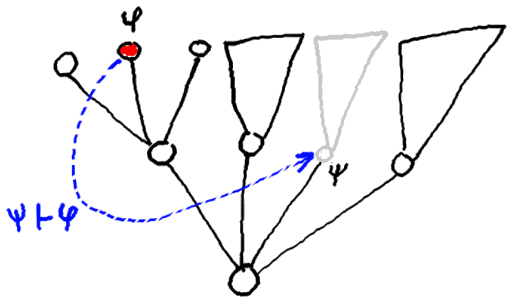


Figure 2. Suppression, *a posteriori*, du nœud ψ (et de ses fils) subsumé par φ

Afin d'aider Cubicle à converger plus rapidement, il est possible d'ajouter des invariants (exprimées à l'aide de *cubes*) qui vont permettre de réduire l'exploration de grandes parties de l'arbre de recherche. Ces propriétés peuvent être ajoutées de deux manières différentes :

– soit comme *candidats invariants*, et elles seront alors *vérifiées* par Cubicle ;

– soit comme *invariants*, et elles seront alors simplement *supposées* par Cubicle. Dans ce cas, la correction du model checker peut être remise en cause si l'invariant fourni est faux.

Cubicle dispose également d'un mécanisme pour synthétiser automatiquement des *candidats* invariants. Chacun de ces candidats est vérifié en exécutant une instance de la boucle de model checking dont les ressources machine sont limitées volontairement (en limitant le nombre de nœuds visités par exemple). En plus de ces invariants synthétisés de manière dynamique, Cubicle découvre des *invariants de sous-typage* à l'aide d'une analyse statique qui calcule, pour chaque variable dont le type est un type énuméré, un sous ensemble des valeurs possibles pour cette variable. Ces invariants sont ensuite envoyés directement au démonstrateur SMT qui supporte la définition de sous-types pour les types énumérés.

Terminaison. La terminaison de l'algorithme d'atteignabilité de Cubicle n'est pas garantie en général, mais on peut montrer qu'elle peut être obtenue sous certaines conditions.

Pour étudier la terminaison de l'algorithme en figure 1, on s'intéresse à l'évolution de l'ensemble V des nœuds visités. Plus précisément, on considère la séquence d'inclusions $V_0 \subseteq V_1 \subseteq \cdots \subseteq V_n \subseteq \cdots$ où V_n représente l'ensemble V à la $n^{\text{ième}}$ itération de la boucle `while`. Pour que la boucle d'atteignabilité ne termine pas, il faut nécessairement que de nouveaux cubes soient ajoutés régulièrement dans V. Par conséquent, il existe une infinité d'ensembles de nœuds visités V_{k_i} tels que $V_{k_1} \subset V_{k_2} \subset \cdots$. On va définir les condiditions suffisantes pour que cette sous-séquence d'inclusions *strictes* soit finie. Le lecteur est renvoyé à [1,17] pour plus de détails sur les propositions et théorèmes suivants.

Étant donné un système paramétré à tableaux $\mathcal{S}$, on appelle *configuration* de $\mathcal{S}$ un état concret du système, c'est-à-dire un modèle pour les types, les variables globales et les tableaux du système. En particulier, une configuration doit fixer le nombre de processus du système (*i.e.* la cardinalité du type `proc`) qui peut être utilisé pour indéxer les tableaux. Un système à tableaux $\mathcal{S}$ a un nombre potentiellement infini de configurations.

Étant donné un cube φ manipulé par Cubicle, on note $[\![\varphi]\!]$ l'ensemble des configurations qui satisfont φ. Si l'ensemble des configurations est muni d'un pré-ordre bien fondé $\preceq$ (*i.e.* une relation binaire réflexive et transitive sans suite infinie strictement décroissante), alors on peut montrer que $[\![\varphi]\!]$ est un *idéal*, c'est-à-dire que si $s \in [\![\varphi]\!]$ et $s \preceq s'$ alors $s' \in [\![\varphi]\!]$. Par extension, il est immédiat de montrer que chaque ensemble V_n de nœuds visités est également un idéal.

Maintenant, si $\preceq$ a également la propriété d'être un *bel ordre*, c'est-à-dire si pour toute séquence infinie de configurations $s_1, s_2, \ldots$, il existe nécessairement $i < j$ tels que $s_i \preceq s_j$, alors la terminaison de l'algorithme est assurée par le théorème suivant :

Theorem 1. *Si $\preceq$ est un bel ordre, alors toute séquence d'inclusions strictes d'idéaux $[\![V_{k_1}]\!] \subset [\![V_{k_2}]\!] \subset \cdots$ est finie.*

De plus, $[\![.]\!]$ est monotone, *i.e.* si $V \subset V'$ alors $[\![V]\!] \subset [\![V']\!]$ donc la finitude de la séquence d'inclusion $[\![V_{k_1}]\!] \subset [\![V_{k_2}]\!] \subset \cdots$ implique bien la finitude de la séquence d'inclusion $V_0 \subseteq V_1 \subseteq \cdots \subseteq V_n \subseteq \cdots$ calculée par l'algorithme. La preuve du théorème 1 repose sur l'argument suivant : par contradiction, si cette séquence est infinie, alors il existe également une séquence infinie de configurations $s_1, s_2, \ldots$ telle que $s_i \in [\![V_{k_i}]\!]$ et $s_i \notin [\![V_{k_j}]\!]$, pour tout $j < i$. De plus, $s_j \not\preceq s_i$, sinon $s_i \in [\![V_{k_j}]\!]$ puisque chaque $[\![V_{k_j}]\!]$ est un idéal. L'existence de cette suite infinie contredit donc l'hypothèse que $\preceq$ est un bel ordre.

Il ne reste donc plus qu'à déterminer les conditions nécessaires pour que l'ensemble des configurations puisse être muni d'un *bel ordre*. Cela dépend essentiellement du choix des opérations de comparaison sur le type `proc` et sur les types des éléments des tableaux.

- Si les éléments des tableaux appartiennent à un type énuméré et que les indices sont seulement munis de la relation d'égalité, alors il est possible d'exhiber un bel-ordre sur les configurations par le lemme de Dickson [17,26].
- Si les éléments des tableaux appartiennent à un type énuméré et que les indices sont munis de la relation d'égalité et d'un ordre total, alors il est possible d'exhiber un bel-ordre sur les configurations par le lemme de Higman [17,26].
- Si les éléments des tableaux sont des rationnels et que les indices sont seulement munis de la relation d'égalité, alors il est possible d'exhiber un bel-ordre sur les configurations par le lemme de Kruskal [17,26].

4 Architecture parallèle

Une manière naturelle d'augmenter la rapidité des model checkers est de paralléliser les tâches qui demandent du temps de calcul pour tirer parti de la disponibilité grandissante des machines multi-cœurs ou multi-processeurs ainsi que des clusters de machines [19,6,24]. Dans le cadre de Cubicle cette parallélisation peut être faite de façon immédiate au niveau de la génération d'invariants car la boucle de model checking qui vérifie ces derniers est complètement indépendante du reste de la recherche. De manière plus intéressante, la boucle d'atteignabilité arrière peut elle même être parallélisée à un certain degré. Une implantation directe d'une boucle parallèle affecterait cependant l'orientation de l'exploration, et casserait les heuristiques employées et pourrait même rendre certaines optimisations de la section 3 non sûres. De plus nos expériences ont montré qu'une exploration non déterministe de l'espace de recherche est souvent moins efficace qu'une recherche guidée.

Dans notre cas, les tâches qui consomment le plus de ressources sont les tests de subsomption (ligne 7 de l'algorithme figure 1) qui peuvent être des problèmes difficiles même pour des démonstrateurs SMT modernes, principalement de par leur taille. Pour paralléliser Cubicle nous avons implanté une version concurrente de la recherche en largeur (BFS) en se reposant sur la simple observation que tous les calculs à faire sur toutes les branches à un même niveau de l'arbre de recherche peuvent être effectués en parallèle. Cette implantation utilise une

architecture centralisée de type maître/esclave. Le maître attribue des tests de point fixe aux esclaves et une barrière de synchronisation est placée à la fin de chaque niveau de l'arbre pour conserver une exploration en largeur. Le maître calcule ensuite de manière asynchrone les pré-images des nœuds qui n'ont pas pu être vérifiés comme étant des points fixes par les esclaves. Pendant ce temps le maître peut aussi attribuer des tâches de génération d'invariants qui seront traitées si des esclaves deviennent disponibles. Maintenant, pour supprimer des nœuds subsumés a posteriori de V de manière à ne pas se retrouver dans le cas de la figure 3, le maître doit simplement ignorer les résultats concernant les nœuds qui ont été supprimés pendant qu'un esclave vérifiait leur propriété de point fixe.

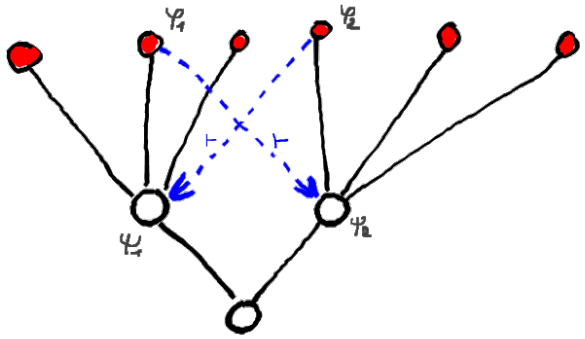


Figure 3. Une mauvaise synchronisation des tests de subsomption effectués en parallèle (flèches en bleu) peut supprimer des nœuds de manière incorrecte (ici, une des deux suppressions par subsomption doit être ignorée pour garantir la sûreté)

D'un point de vue technique, Cubicle fournit une exploration parallèle de l'espace de recherche en utilisant n processus concurrents sur une architecture multi-cœurs lorsqu'il est invoqué avec l'option `-j` n. L'implantation utilise Functory [15], une bibliothèque OCaml fournissant une interface fonctionnelle riche et polymorphe et permet facilement de distribuer des calculs parallèles. Functory permet d'utiliser des architectures multi-cœurs ainsi que des réseaux de machines et fournit un mécanisme robuste de tolérance aux pannes. Bien que Cubicle ne fournisse pas aujourd'hui d'implantation distribuée, c'est une des directions envisagées pour l'évolution du logiciel qui demanderait tout de même d'être capable de limiter la taille des données devant circuler sur le réseau lors des communications entre le maître et les esclaves. Ici c'est la taille des nœuds visités V qui peut rapidement devenir un goulot d'étranglement dans une architecture distribuée reposant sur l'échange de messages. Une solution à ce problème est que chaque esclave maintienne sa propre copie de V et que seules les mises à jour de cet ensemble soient transmises.

5 La bibliothèque Alt-Ergo zero

Cubicle se sert de manière intensive d'un démonstrateur SMT pour décharger ses tests de sûreté et de subsomption. Pour cela, il est distribué avec son propre démonstrateur qu'il invoque, automatiquement, au travers d'une API OCaml. Ce démonstrateur, dérivé d'Alt-Ergo et baptisé Alt-Ergo zero, est aussi distribué sous la forme d'une bibliothèque OCaml disponible librement sur le web à l'adresse `http://cubicle.lri.fr/alt-ergo-zero/`. La documentation complète au format Ocamldoc est disponible sur le web à la même adresse. Nous décrivons ici brièvement l'utilisation de cette bibliothèque.

Alt-Ergo zero prend en entrée des formules logiques sans quantificateurs combinant des opérateurs de la logique propositionnelle avec des opérateurs prédéfinis pour les théories de l'égalité, de l'arithmétique linéaire (entiers et rationnels) et des types énumérés.

Pour des raisons d'efficacité, Alt-Ergo zero utilise des chaînes de caractères hash-consées [12] pour un partage maximal (et une opération de comparaison efficace), dont l'utilisation est aussi documentée mais ne sera pas décrite dans cette section. L'API du démonstrateur se divise en six modules qui fournissent des fonctionnalités de typage (`Type`, `Symbol` et `Variant`), de construction de termes (`Term`) et de formules (`Formula`), et d'appel au démonstrateur (`Solver`).

5.1 Typage

Les types de base des entiers, des réels, des booléens et des processus sont fournis directement, mais il est toujours possible de déclarer un nouveau type abstrait en appelant la fonction `Type.declare "mon_type" []` ou de définir un type énuméré ayant comme constructeurs `A` et `B` en appelant `Type.declare "mon_type" ["A"; "B"]`. Le module `Type` permet aussi de récupérer les constructeurs d'un type.

```
module Type : sig
  type t = Hstring.t

  val type_int : t
  val type_real : t
  val type_bool : t
  val type_proc : t

  val declare : Hstring.t -> Hstring.t list -> unit
  val constructors : t -> Hstring.t list
  ...
end
```

Le module `Symbol` suivant permet quant à lui de déclarer des symboles de fonction qui, s'ils ne prennent aucun argument en entrée seront des constantes. Le

deuxième argument de `Symbol.declare` est la liste des types de ses arguments et le troisième est son type de retour. Ce module permet aussi de faire différentes requêtes sur les types des symboles.

```
module Symbol : sig
  type t = Hstring.t

  val declare : Hstring.t -> Type.t list -> Type.t -> unit
  val type_of : t -> Type.t list * Type.t
  val has_abstract_type : t -> bool
  val has_type_proc : t -> bool
  val declared : t -> bool
end
```

Enfin le module `Variant` permet de faire une analyse de sous-typage. Pour utiliser cette fonctionnalité, il faut tout d'abord initialiser les types des symboles avec la fonction `Variant.init`, puis ajouter une à une les contraintes de sous typage que l'on désire voir respectées. Ensuite, un seul appel à la fonction `Variant.close` suffit à calculer les types les plus petits possibles et à mettre à jour l'information apportée par ces raffinements dans l'environnement de typage global.

```
module Variant : sig
  val init : (Symbol.t * Type.t) list -> unit
  val close : unit -> unit
  val assign_constr : Symbol.t -> Hstring.t -> unit
  val assign_var : Symbol.t -> Symbol.t -> unit
  ...
end
```

5.2 Construction de termes et formules

Pour construire des termes arithmétiques, Alt-Ergo ZERO fournit les opérateurs classiques ainsi que des fonctions pour créer des constantes numériques et des applications de fonction.

```
module Term : sig
  type t
  type operator = Plus | Minus | Mult | Div | Modulo

  val make_int : Num.num -> t
  val make_real : Num.num -> t
  val make_app : Symbol.t -> t list -> t
  val make_arith : operator -> t -> t -> t

  val is_int : t -> bool
  val is_real : t -> bool

  val t_true : t
  val t_false : t
end
```

Les formules sont soit des littéraux construits avec la fonction `make_lit` soit des combinaisons de littéraux construits avec la fonction `make`.

```
module Formula : sig
  type comparator = Eq  | Neq | Le  | Lt
  type combinator = And | Or  | Imp | Not
  type t =
    | Lit of Literal.LT.t
    | Comb of combinator * t list

  val f_true : t
  val f_false : t

  val make_lit : comparator -> Term.t list -> t
  val make : combinator -> t list -> t
  ...
end
```

5.3 Utiliser le démonstateur SMT

L'interface du démonstrateur d'Alt-Ergo ZERO est donnée dans la figure 4. Elle se présente comme un foncteur `Make` qui prend en argument un module avec une signature vide. Cette interface fonctorisée est utilisée pour permettre la création de plusieurs instances de démonstrateur. Chaque démonstrateur obtenu par l'application de ce foncteur est impératif. L'état interne est représenté par un type abstrait et deux fonctions `save_state` et `restore_state` qui permettent respectivement de sauvegarder et de restaurer cet état à volonté. La

fonction `assume` permet d'ajouter une formule au contexte et de lui donner un identifiant. La mise en forme normale conjonctive est faite à ce moment et les propagations unitaires possibles sont effectuées. Il est possible à tout moment de vérifier le contexte du démonstrateur en appelant la fonction `check` ce qui lancera réellement Alt-Ergo ZERO. Si le contexte du démonstrateur est incohérent alors l'exception `Unsat` est levée, accompagnée d'une liste d'entiers correspondant au *unsat core* (ou noyau d'insatisfiabilité) sous la forme d'identifiants associés aux formules supposées avec la fonction `assume`. Cette interface fournit aussi une fonction `entails` pour vérifier si le contexte du démonstrateur implique une formule donnée, sans en changer l'état interne.

```
exception Unsat of int list

module type Solver = sig
  type state
  ...
  val clear : unit -> unit
  val save_state : unit -> state
  val restore_state : state -> unit

  val assume : ?profiling:bool -> id:int -> Formula.t -> unit
  val check : ?profiling:bool -> unit -> unit
  val entails : ?profiling:bool -> id:int -> Formula.t -> bool
end

module Make (Dummy : sig end) : Solver
```

Figure 4. Interface du démonstrateur d'Alt-Ergo ZERO

L'exemple ci-dessous montre comment utiliser Alt-Ergo ZERO pour déterminer la (non) satisfiabilité de la conjonction de littéraux suivante :

$$f(x+3) = u \wedge f(y+2) = w \wedge x = y - 1 \wedge u \neq w$$

où `t` est un type abstrait, f un symbole de fonction de type `int` $\rightarrow$ `t`, x et y deux entiers et u et w deux variables de type `t`

On commence par créer une instance du démonstrateur (et quelques raccourcis pour gagner en visibilité) de la manière suivante :

```
open Smt
module S = Symbol
module T = Term
module F = Formula
module Solver = Make (struct end)
```

Puis, on déclare le type abstrait `t` à l'aide de la fonction `Type.declare`. Le nom du type est simplement défini à l'aide d'une chaîne *hash consées* `"t"` :

```
let type_t = Type.declare (Hstring.make "t") []
```

Ensuite, on déclare les symboles de la formule à l'aide de la fonction `declare` du module `Symbol` :

```
let x = S.declare (Hstring.make "x") [] type_int
let y = S.declare (Hstring.make "y") [] type_int
let u = S.declare (Hstring.make "u") [] type_t
let w = S.declare (Hstring.make "w") [] type_t
let f = S.declare (Hstring.make "f") [Type.type_int] type_t
```

Enfin, chaque sous-terme de la formule est défini à l'aide des fonctions `make_app`, `make_int` et `make_arith` du module `Term` :

```
let tx = T.make_app x []
let ty = T.make_app y []
let tu = T.make_app u []
let tw = T.make_app w []
let t3 = T.make_int (Num.Int 3) []
let t2 = T.make_int (Num.Int 2) []
let t1 = T.make_int (Num.Int 1) []
let fx3 = T.make_app f (T.make_arith T.Plus tx t3)
let fy2 = T.make_app f (T.make_arith T.Plus ty t2)
```

Chaque littéral est ensuite défini à l'aide de la fonction `Formula.make_lit` de la manière suivante :

```
let l1 = F.make_lit F.Eq [fx3; tu]            (* f(x + 3) = u *)
let l2 = F.make_lit F.Eq [fy2; tw]            (* f(y + 2) = w *)
let l3 =
   F.make_lit F.Eq
        [tx; (T.make_arith T.Minus ty t1)]    (* x = y - 1 *)
let neg_goal = F.make_lit F.Neq [tu; tw]      (* u <> w *)
```

Les littéraux sont enfin ajoutés les uns après les autres au contexte de l'instance du démonstrateur à l'aide de la fonction `Solver.assume`. Puis on vérifie la satisfiabilité du contexte final en appelant la fonction `Solver.check` :

```
try
  Solver.clear ();
  Solver.assume ~id:1 l1;
  Solver.assume ~id:2 l2;
  Solver.assume ~id:3 l3;
  Solver.assume ~id:4 neg_goal;
  Solver.check ();
  print_endline "satisfiable"
with Unsat _ -> print_endline "unsatisfiable"
```

6 Expériences

Nous avons évalué Cubicle sur des algorithmes d'exclusion mutuelle et des protocoles de cohérence de cache classiques mais aussi sur des protocoles plus difficiles à prouver (grand nombre de transitions, de variables, etc.). Dans le tableau figure 5, on compare les performances de Cubicle avec d'autres model checkers existants pour systèmes paramétrés. Toutes les expériences ont été réalisées sur une machine 64 bits avec un processeur quadri-cœurs Intel® Xeon® cadencé à 3,2 GHz et comportant 24 Go de mémoire. Pour chaque outil, on donne les résultats obtenus avec les meilleurs réglages qu'on ait trouvé. Il est à noter que la version concurrente de Cubicle a été lancée sur quatre cœurs (i.e. avec l'option `-j 4`). Cette version n'a été exécutée que sur les exemples qui prennaient un temps significatif (> 10 secondes) en séquentiel. On a marqué par X les *benchmarks* qu'il nous a été impossible de traduire à cause de restrictions syntaxiques.

	Cubicle		MCMT [18]	Undip [3]	PFS [2]
	seq.	4 cœurs			
bakery	0.01s	–	0.01s	0.04s	0.01s
Dijkstra	0.24s	–	0.99s	**0.04s**	0.26s
Distributed_Lamport	**2.3s**	–	12.7s	unsafe	X
Java_Mlock	0.04s	–	0.06s	0.25s	0.02s
Ricart_Agrawala	**1.8s**	–	1m12s	4.3s	X
Szymanski_at	**0.12s**	–	0.71s	13.5s	timeout
Berkeley	0.01s	–	0.01s	0.01s	0.01s
flash_aggregated [25]	0.01s	–	0.02s	0.01s	X
German_Baukus	**25.0s**	17.1s	3h39m	9m43s	X
German_pfs	**6m23s**	3m8s	11m31s	timeout	47m22s
German_undip	**0.17s**	–	0.57s	1m32	X
Illinois	0.02s	–	0.04s	0.06s	0.06s
Moesi	0.01s	–	0.01s	0.01s	0.01s

Figure 5. Benchmarks

Ces résultats sont très prometteurs. En effet, ils montrent tout d'abord que la version séquentielle de Cubicle est compétitive et que la version parallèle est capable d'atteindre une accélération d'environ $1,8$ sur quatre cœurs. C'est un bon résultat si on considère que toutes les unités de calcul ne peuvent pas être utilisées à leur maximum à cause des barrières de synchronisation requises pour garder une stratégie pertinente. En pratique, on a remarqué que les meilleures performances étaient obtenues en utilisant toutes les optimisations décrites dans la section 3 (à l'exception de la génération d'invariants qui peut demander beaucoup de temps pour des résultats incertains). Dans le tableau figure 6, on met en évidence les effets des différentes optimisations sur une version du protocole German extrait de [7]. La colonne "permut." désigne le calcul des permutations pertinentes et "suppression" désigne la suppression des nœuds subsumés *a pos-*

teriori. En particulier il est intéressant de remarquer que l'analyse statique de sous-typage améliore les performances d'un ordre de grandeur sur cet exemple.

Optimisations				**Temps réel** (nb. de nœuds)	
permut.	suppression	sous-typage	invariants	séquentiel	4 cœurs
Non	Non	Non	Non	∞	∞
Oui	Non	Non	Non	50m8s (22580)	27m13s (20710)
Oui	**Oui**	Non	Non	35m16s (20405)	19m39s (19685)
Oui	**Oui**	Non	**Oui**	20m45s (15089)	13m55s (14527)
Oui	**Oui**	**Oui**	Non	**25.0s** (3322)	**17.1s** (3188)

Figure 6. Effets des différentes optimisations sur le protocole German

Le code Cubicle complet du protocole German est donné dans l'annexe A.

7 Conclusion et perspectives

Nous avons présenté Cubicle, un model checker capable de prouver des propriétés de sûreté de systèmes de transition paramétrés. Son langage d'entrée permet notamment de décrire des algorithmes d'exclusion mutuelle et des protocoles de cohérence de cache intéressants comme le German ou le FLASH [23]. Les expériences menées montrent que Cubicle est très compétitif par rapport aux model checkers de la même famille. Cubicle utilise la bibliothèque Functory pour son architecture parallèle et la bibliothèque SMT Alt-Ergo zero, une version allégée et améliorée d'Alt-Ergo.

Les perspectives envisagées pour l'évolution de Cubicle concernent à la fois son expressivité et son efficacité. En terme d'expressivité, nous souhaitons étendre le langage d'entrée avec tout d'abord, un langage procédural de plus haut niveau (fonctions, séquences, boucles, primitives de communications, threads). Ensuite, nous envisageons une extension du langage logique pour décrire plus largement les formules définissant les états initiaux. Enfin la définition des types peut être étendue avec des structures de données comme les enregistrements ou les types sommes. Pour réaliser cela, le démonstrateur Alt-Ergo zero devra aussi intégrer de nouvelles procédures de décision.

Concernant l'efficacité de Cubicle, le prochain défi est la preuve de sûreté du protocole FLASH [23], considéré comme un des plus complexes protocoles de cohérence de cache académiques. Il comporte 600 millions d'états accessibles lorsque seulement quatre processus sont mis en jeu. Encore aujourd'hui, peu de méthodes sont capables de prouver la sûreté d'un tel protocole et toutes requièrent une intervention humaine [9,27]. En comparaison, le protocole German compte 40 000 états pour quatre processus. Pour faire la preuve de sûreté de FLASH, et donc espérer pouvoir aborder des protocoles de taille industrielle, il

est nécessaire de chercher des techniques pour réduire l'espace d'états de Cubicle, et améliorer son mécanisme de génération d'invariants. Ceci fait l'objet d'un travail en cours.

La bibliothèque OCaml Alt-Ergo ZERO a été conçue pour être autonome et utilisable dans d'autres contextes. Par exemple, on envisage de l'utiliser pour développer un autre type de model checker par k-induction et également de s'en servir dans des outils de test. Pour cela Alt-Ergo ZERO doit être étendu pour renvoyer des modèles servant à construire des traces et contre-exemples avec des variables instanciées.

Références

1. Parosh Aziz Abdulla, Karlis Cerans, Bengt Jonsson, and Yih-Kuen Tsay. General decidability theorems for infinite-state systems. In *LICS*, pages 313–321, 1996.
2. Parosh Aziz Abdulla, Giorgio Delzanno, Noomene Ben Henda, and Ahmed Rezine. Regular model checking without transducers (On efficient verification of parameterized systems). In *TACAS*, pages 721–736, 2007.
3. Parosh Aziz Abdulla, Giorgio Delzanno, and Ahmed Rezine. Parameterized verification of infinite-state processes with global conditions. In *CAV*, pages 145–157, 2007.
4. Aurore Annichini, Ahmed Bouajjani, and Mihaela Sighireanu. TReX : A tool for reachability analysis of complex systems. In *CAV*, pages 368–372. 2001.
5. Krzysztof Apt and Dexter Kozen. Limits for automatic verification of finite-state concurrent systems. *Information Processing Letters*, 22 :307–309, 1986.
6. J. Barnat, L. Brim, M. Češka, and P. Ročkai. DiVinE : Parallel Distributed Model Checker (Tool paper). In *HiBi/PDMC*, pages 4–7, 2010.
7. Kai Baukus, Yassine Lakhnech, and Karsten Stahl. Parameterized verification of a cache coherence protocol : Safety and liveness. In *VMCAI*, pages 317–330, 2002.
8. Randal E. Bryant, Shuvendu K. Lahiri, and Sanjit A. Seshia. Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In *CAV*, pages 78–92, 2002.
9. Ching-Tsun Chou, Phanindra Mannava, and Seungjoon Park. A simple method for parameterized verification of cache coherence protocols. In *FMCAD*, pages 382–398. 2004.
10. Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. Nusmv 2 : An opensource tool for symbolic model checking. In *CAV*, pages 241–268. 2002.
11. Sylvain Conchon, Evelyne Contejean, Johannes Kanig, and Stéphane Lescuyer. CC(X) : Semantic combination of congruence closure with solvable theories. *ENTCS*, 198(2) :51–69, 2008.
12. Sylvain Conchon and Jean-Christophe Filliâtre. Type-Safe Modular Hash-Consing. In *ACM SIGPLAN Workshop on ML*, Portland, Oregon, September 2006.
13. Sylvain Conchon, Amit Goel, Sava Krstić, Alain Mebsout, and Fatiha Zaïdi. Cubicle : A parallel SMT-based model checker for parameterized systems. In *CAV*, pages 718–724. 2012.

14. David L. Dill, Andreas J. Drexler, Alan J. Hu, and C. Han Yang. Protocol verification as a hardware design aid. In *ICCD*, pages 522–525, 1992.

15. Jean-Christophe Filliâtre and K. Kalyanasundaram. Functory : A distributed computing library for Objective Caml. In *TFP*, pages 65–81, 2011.

16. Silvio Ghilardi, Enrica Nicolini, Silvio Ranise, and Daniele Zucchelli. Towards SMT model checking of array-based systems. In *IJCAR*, pages 67–82, 2008.

17. Silvio Ghilardi and Silvio Ranise. Backward reachability of array-based systems by SMT solving : Termination and invariant synthesis. *LMCS*, 6(4), 2010.

18. Silvio Ghilardi and Silvio Ranise. MCMT : A model checker modulo theories. In *IJCAR*, pages 22–29, 2010.

19. Orna Grumberg, Tamir Heyman, Nili Ifergan, and Assaf Schuster. Achieving speedups in distributed symbolic reachability analysis through asynchronous computation. In *CHARME*, pages 129–145, 2005.

20. Klaus Havelund and Thomas Pressburger. Model checking java programs using java pathfinder. *International Journal on Software Tools for Technology Transfer (STTT)*, 2 :366–381, 2000.

21. Thomas Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Software verification with blast. In *Model Checking Software*, pages 624–624. 2003.

22. G.J. Holzmann. The model checker spin. *Software Engineering, IEEE Transactions on*, 23(5) :279 –295, may 1997.

23. J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The Stanford FLASH multiprocessor. In *Computer Architecture*, pages 302 –313, apr 1994.

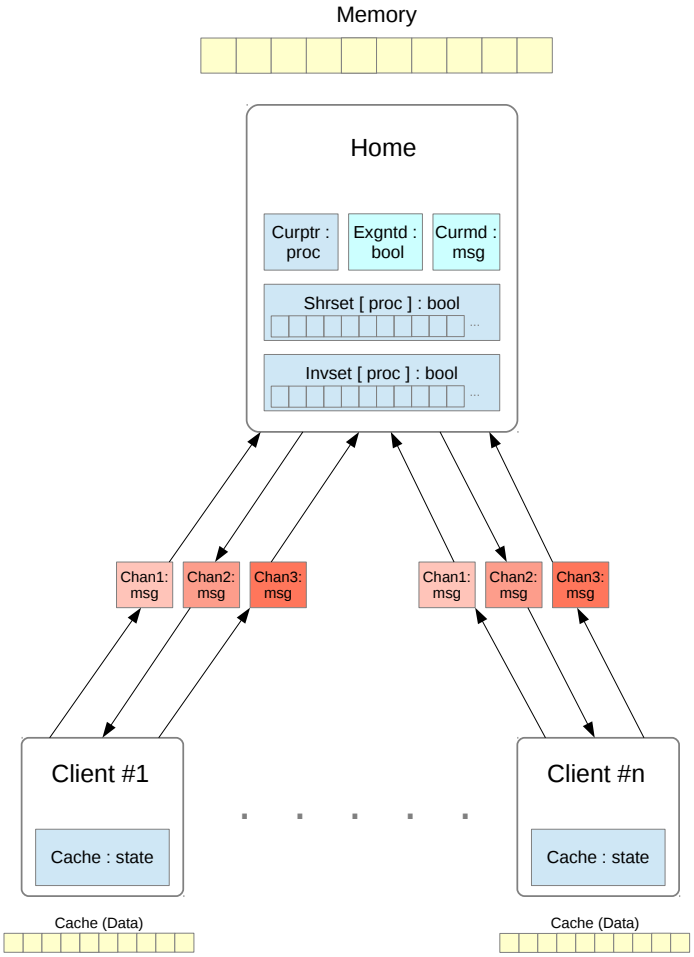
24. Igor Melatti, Robert Palmer, Geoffrey Sawaya, Yu Yang, Robert M. Kirby, and Ganesh Gopalakrishnan. Parallel and distributed model checking in Eddy. *STTT*, 11(1) :13–25, 2009.

25. Seungjoon Park and David L. Dill. Protocol verification by aggregation of distributed transactions. In *CAV*, pages 300–310, 1996.

26. Sylvain Schmitz and Philippe Schnoebolen. Algorithmic Aspects of WQO (Well-Quasy-Ordering) Theory. ESSLLI, 2012.

27. Murali Talupur and Mark R. Tuttle. Going with the flow : parameterized verification using message flows. In *FMCAD*, pages 10 :1–10 :8, Piscataway, NJ, USA, 2008. IEEE Press.

A Le protocole German



german.cub

```
type state = Invalid | Shared | Exclusive
type msg = Empty | Reqs | Reqe | Inv | Invack | Gnts | Gnte

var Exgntd : bool
var Curcmd : msg
var CurClient : proc

array Chan1[proc] : msg
array Chan2[proc] : msg
array Chan3[proc] : msg
```

```
array Cache[proc] : state
array Invset[proc] : bool
array Shrset[proc] : bool

init (z) {
  Cache[z] = Invalid && Chan1[z] = Empty &&
  Chan2[z] = Empty && Chan3[z] = Empty &&
  Invset[z] = False && Shrset[z] = False &&
  Curcmd=Empty && Exgntd = False }

unsafe (z1 z2)
  { Cache[z1] = Exclusive && Cache[z2] <> Invalid }

transition send_req_shared(n)
requires { Cache[n] = Invalid && Chan1[n] = Empty }
{ Chan1[n] := Reqs; }

   transition send_req_exclusive_1(n)
   requires { Cache[n] = Invalid && Chan1[n] = Empty }
   { Chan1[n] := Reqe; }

   transition send_req_exclusive_2(n)
   requires { Cache[n] = Shared && Chan1[n] = Empty }
   { Chan1[n] := Reqe; }

   transition recv_req_shared(n)
   requires { Curcmd = Empty && Chan1[n] = Reqs }
   { Curcmd := Reqs;
     CurClient := n;
     Invset[j] := case | _ : Shrset[j];
     Chan1[n] := Empty; }

   transition recv_req_exclusive(n)
   requires { Curcmd = Empty && Chan1[n] = Reqe }
   { Curcmd := Reqe;
     CurClient := n;
     Invset[j] := case | _ : Shrset[j];
     Chan1[n] := Empty; }

   transition send_inv_1(n)
   requires { Chan2[n] = Empty && Invset[n] = True &&
              Curcmd = Reqe }
   { Chan2[n] := Inv;
     Invset[n] := False; }
```

```
transition send_inv_2(n)
requires { Chan2[n] = Empty && Invset[n] = True &&
           Curcmd = Reqs && Exgntd = True}
{ Chan2[n] := Inv;
  Invset[n] := False; }

transition send_invack(n)
requires { Chan2[n] = Inv && Chan3[n] = Empty }
{ Chan2[n] := Empty;
  Chan3[n] := Invack;
  Cache[n] := Invalid; }

transition recv_invack(n)
requires { Chan3[n] = Invack && Curcmd <> Empty }
{ Exgntd := False;
  Chan3[n] := Empty;
  Shrset[n] := False; }

transition send_gnt_shared(n)
requires { CurClient = n && Curcmd = Reqs &&
           Exgntd = False && Chan2[n] = Empty }
{ Curcmd := Empty;
  Chan2[n] := Gnts;
  Shrset[n] := True; }

transition send_gnt_exclusive(n)
requires { CurClient = n && Curcmd = Reqe &&
           Chan2[n] = Empty && Shrset[n] = False &&
           forall_other j. Shrset[j] = False }
{ Curcmd := Empty;
  Exgntd := True ;
  Chan2[n] := Gnte;
  Shrset[n] := True; }

transition recv_gnt_shared(n)
requires { Chan2[n] = Gnts }
{ Cache[n] := Shared;
  Chan2[n] := Empty; }

transition recv_gnt_exclusive(n)
requires { Chan2[n] = Gnte }
{ Cache[n] := Exclusive;
  Chan2[n] := Empty; }
```

Wave analysis of Advanced Self-Modifying Behaviors

Jean-Yves Marion, Daniel Reynaud

Nancy University - Loria
Berkeley University
Email: Jean-Yves.Marion@loria.fr, danielreynaud@cs.berkeley.edu

Abstract. A common code protection technique to evade antivirus detection consists in packing, that is to say turning any program into a self-modifying program. As a result, most malware samples in the wild are packed, increasing the vulnerability window for systems protected by signature-based security solutions.
While unpacking has already been researched, we propose a dynamic typing framework to accurately classify different forms of self-modifications and expose code waves (i.e. nested layers of self-modifying code).
Our contributions are a taxonomy of self-modifying behaviors, a detection of non-interference properties between code waves and a new visualization for self-modifying programs that could be used as a signature.

1 Introduction

1.1 Motivation

We study self-modifying programs from both a theoretical and a practical perspective. This research originated in malware analysis and in particular the analysis of code armoring techniques, but the resulting framework is very general and has many other potential applications. There are many reasons to study self-modifying programs, including:

- Semantics of self-modifying programs are rather unclear. To our knowledge, only a few papers suggest such semantics [9,6,26,5]. It seems to be a challenging direction.
- From a computability point of view, self-reproduction and self-reference are particular cases of self-modifying programs [5]. Kleene's second recursion theorem [20] plays a key role for sequential computational models.
- Manual analysis and static analysis of self-modifying programs are difficult, because the program text is not available due to rewriting or misalignment.
- Runtime analysis is currently the only practical approach. For example, approaches based on dynamic tainting [34,36] give good results. However, there is a need to move to predictive analysis in order to guarantee some safety properties.

– From a practical point of view, self-modifying programs are used for dynamic program optimization, just-in-time compilation, code protection and compression.
– Self-modifying methods are extensively and efficiently used by malware to avoid anti-virus detection.

1.2 Results

In this paper, we address the problem of automatic analysis of self-modifying programs, with a focus on dynamically protected programs such as malware. In order to model self-modifying programs, we introduce the notion of pseudo-programs, for which the program text is not fixed w.r.t. semantics. We then develop a type system which collects information at runtime (like tainting), but which also has the ability to predict information-flow properties (like traditional type systems). The key idea is to type memory locations at runtime with an execution level thanks to dynamic binary instrumentation or emulation. Basically, an instruction has level $k + 1$ if it was written by an instruction at level k. This leads us to explain a self-modifying program execution as a sequence of code waves, each wave being the set of instructions with level k. Next, we use this typing information to define behavior patterns, which give a high level description of decrypted or scrambled code for example. With these behavior patterns we are able to classify binaries, to visualize their temporal evolution and to detect suspect runs.

The dynamic typing system can be directly used to:

– classify binaries w.r.t. behavior patterns
– determine code boundaries and the cleartext form of self-modifying code
– extract behavioral signatures (memory-wise) as a graph
– raise alerts when suspicious runs are detected
– find non-interference properties between code waves

1.3 Related Work

The method described in this paper is a generalization of a usual technique for automatic unpacking, described thoroughly in the literature. This technique consists generally in emulating the execution of the application, generating a set of instruction pointers and a set of written addresses, and then computing the intersection between these sets. If this intersection is not empty, it is assumed to be unpacked code, so the memory is dumped and the reconstruction of an unpacked executable can be attempted from the memory dump. Numerous implementations have been based on this model, including Renovo [19], Saffron [27], OmniUnpack [23], Pandora's Bochs [4], Ether [10]...

Other approaches such as PolyUnpack [28] or VxStripper [18] are conceptually different from the method described here because they are based on the comparison between runtime data and statically accessible data.

Unpackers such as OllyBonE [33] or the Universal PE Unpacker [8] can't be compared with our method because they only provide semi-automatic unpacking or rely on system-specific heuristics such as the use of some API calls such as GetProcAddress on Windows systems.

Finally, some unpackers work either statically [17] or semi-statically [21]. This method requires identifying a known packer using a signature or a known code chunk and calling a specific unpacking procedure based on this identification. Again, this method can't be compared with ours since it requires the development of specific procedures for each packer.

A number of technical counter-measures can be found against automatic unpackers, such as anti-emulation and anti-dumping techniques [12]. Other more fundamental techniques can be found such as using dual-mapping (referring to the same physical address with different virtual addresses) [32] or bytecode interpreters.

Note that although we build on the technique for automatic unpacking, the output of the dynamic typing system (a multigraph representing the whole trace) can not be compared with the output of automatic unpackers (a program dumped after a linear process). It should be noted however that our typing system, giving a clear definition of code waves boundaries, could be integrated into existing unpackers.

1.4 Comparison with prior work

The most common tools for self-modifying code analysis are unpackers. The purpose of the framework described in this paper is not to provide yet another unpacker. We rather try to:

- explore advanced self-modification techniques (used in malware, but also in Digital Rights Management systems and JIT compilers)
- provide formal definitions of code layers and behavior patterns (to the best of our knowledge, this has not been done before). For instance, we provide a rigorous definition of code integrity checking.

Therefore, the techniques described below are not meant to compete with existing automatic unpackers but rather to complement them. Since a formal definition of behavior patterns based on simple instruction semantics is given, our results can be freely reproduced by independent third parties. See section 1.3 for a detailed discussion of related work.

In [15], we described the implementation details of a prototype called TraceSurfer and performed a large-scale experiment on malware samples collected by a honeypot of the High Security lab at Loria & INRIA. We focused on the experimental setup on clusters and other anti-analysis techniques found in the wild. In contrast, the current study sets the scientific foundations of trace-based semantics of self-modifying programs, with a focus on the applications in other fields. The results we introduce are therefore independent on any specific implementation.

1.5 An example of self-modifying code

We will introduce our ideas with a simple self-modifying program in pseudo-assembly code. It is a simplified version of the decryption loop in the Parite.B virus (see figure 1).

```
@a: mov esi, $index
@b: xor [@offset + esi], $key
@c: sub esi, 4
@d: jnz @b
@offset: [encrypted data]
```

Fig. 1. Decryption loop example

We are going to follow the execution of the decryption loop instruction by instruction in order to show how we deal with dynamic code:

1. We execute the first instruction, at address @a. Since it was executed, we assign an execution value of 1 to address @a.
2. We then execute the instruction at address @b, therefore @b has an execution level of 1. Since this instruction both reads and writes the content of memory address @offset+esi, this memory address now has a read and a write level of 1 (the execution level of @b).
3. We execute the instructions at address @c and @d, they both have an execution level of 1.
4. If the @d branch is taken, we execute the loop again. The levels already assigned will not change.
5. When the loop is over, the branch is not taken and the instruction at @offset is executed. Since this instruction already has a write level of 1 (it has been written by the xor instruction), it now has an execution level of 2. We have found the first layer of dynamic code!

Now suppose that one of the instructions with an execution level of 2 also reads and writes memory, it will propagate its execution level to the memory addresses it touches.

Finally, suppose that one of the memory addresses with a write level of 2 is executed, it will have an execution level of 3. This would be the second layer of dynamic code.

This model is simple but allows to track precisely different layers of dynamic code. Indeed, we can observe that:

- memory addresses with an execution level of 0 have never been executed
- memory addresses with an execution level of 1 may have been executed an arbitrary number of times but have not been written by the program itself (i.e. not self-modifying code)

– memory addresses with an execution level of $n+1$ have been written by the program itself, and more precisely by code at level of n

We will show that by correlating this information with the read and write levels, we obtain a taxonomy of self-modifying programs.

1.6 Roadmap

We think that in order to build a robust application, we need a sound theoretical background. For this reason, the first half of this paper discusses theoretical aspects, and the second half focuses more on practical issues.

We first define semantics for pseudo-programs in section 2, which is intuitively a self-modifying program. We suggest a dynamic type system in section 3.

We then use this type system in section 4 to define different behavior (i.e. code protection) patterns and we prove a weak non-interference property based on manadatory access control.

Then the applications and pointers to experiments are detailed in section 5.

2 Self-Modifying Programs

2.1 An operational semantics of pseudo-programs

The memory is organized as a sequence of 8-bit bytes. The set of bytes is noted Bytes. We consider a flat memory model and the set of addresses (the address space) is noted Words. Typically, the address space is in the range $0..2^{32}-1$, that is 4 Bytes. The memory is represented as a finite mapping μ : Words $\rightarrow$ Bytes.

Registers are represented by a finite function ν : Registers $\rightarrow$ Words, where Registers is the set of registers for a given architecture. We note `ip` the instruction pointer register.

The execution environment is a couple (μ, ν) which represents a state of the system. The next execution environment is given by the instruction pointed by `ip`, which is $\mu(\nu(\texttt{ip}))$. An execution is a sequence of execution environments:
$(\mu_0, \nu_0) \rightarrow (\mu_1, \nu_1) \rightarrow \ldots \rightarrow (\mu_n, \nu_n) \rightarrow \ldots$

This sequence may be finite if the computation ends or infinite. The initial execution environment is (μ_0, ν_0) and gives memory and register values at the beginning of the run.

A single transition $(\mu_k, \nu_k) \rightarrow (\mu_{k+1}, \nu_{k+1})$ is completely determined by the execution environment (μ_k, ν_k). The instruction `mov [400],eax`, for example, takes the content $\nu_k(\texttt{eax})$ of the register `eax` and puts it into the memory location 400, that is $\mu_{k+1}(400) = \nu_k(\texttt{eax})$. Here, we transfer $4 - $ Bytes of `eax` into memory. In other words, we write $\mu(m) = x$ to mean the transfer of all the bytes of x (source) at a time into the memory location (destination). We shall use this convention all along. Moreover, we shall not give the formal transition rule for each instruction, which would be tedious and outside the scope of this paper.

When we look at classical textbooks, like [16,35,24], most program semantics consider programs as static objects. A *program* has a fixed text. Its code is the (compiled) list of instructions given by the program text. The code is available and invariant with regard to any execution. In the context of self-modifying code and code misalignment, the situation is different. For this reason, we introduce *pseudo-programs*. Given a semantics, a *pseudo-program* is defined by a pseudo-code, which is a finite sequence of bytes, that is a word of $\textsc{Bytes}^*$. The code of a pseudo-program is inside this pseudo-code. The introductory example is a typical case of a pseudo-program. This definition is justified by the fact that we may not be able to identify instructions inside data. It is worth noting that separating code from data is an undecidable property because we can not always predict the interpretation of instructions like `jmp eax`. As a consequence, the code of a pseudo-program is not always available. At a given step during the execution, we can only see a fragment of the code. So we will speak of *code waves* to describe the part of a program that we can observe at some point of its execution. The definition of code waves will be given in section 3. Notice also that we do not expect pseudo-programs to halt.

The lightweight notion of pseudo-programs w.r.t. semantics that we presented is very convenient because it makes no distinction between code and data. So, it allows to have operational semantics dealing with both self-modifying and misaligned code.

2.2 A stratified dynamic typing system

Each memory address can be read (R), written (W) or executed (X).

We associate to each address m a read level, a write level and an execution level. For this, we define formally a typing environment, which is a mapping $\Gamma : \textsc{Words} \to \mathbb{N}^3$ with the three projections R, W and X.

If $\Gamma(m) = (k_r, k_w, k_x)$, then we say that

1. the reading level of m is k_r w.r.t. Γ and we write $\Gamma \vdash m : R(k_r)$,
2. the writing level of m is k_w w.r.t. Γ and we write $\Gamma \vdash m : W(k_w)$,
3. the executing level of m is k_x w.r.t. Γ and we write $\Gamma \vdash m : X(k_x)$.

A typed execution environment is a triple (μ, ν, Γ) where (μ, ν) is an execution environment and Γ is a typing environment. the execution level of an execution environment is given by the execution level of the address pointed by the `ip` register. That is the execution level of (μ, ν) is k if $\Gamma \vdash \nu(\texttt{ip}) : X(k)$.

Consider the typed execution environment (μ, ν, Γ). We have

$$(\mu, \nu, \Gamma) \to (\mu', \nu', \Gamma')$$

if $(\mu, \nu) \to (\mu', \nu')$ and if Γ' is obtained from Γ by first applying the execution rule:

– *Execution Rule:*

$$\Gamma'(m') = \begin{cases} (k_r, k, k+1) & \text{if } m' = \nu(\texttt{ip}) \text{ and} \\ & \Gamma(\nu(\texttt{ip})) = (k_r, k, k_x) \\ \Gamma(m') & \text{otherwise} \end{cases}$$

This means that code written at level k has an execution level of $k+1$.

Then, we apply the rules below according to the instruction pointed by `ip`. We suppose that $k+1$ is the execution level given by $\nu(\texttt{ip})$.

- *Memory Read Rule:* if $\nu(\texttt{ip})$ points to an instruction which reads memory at address m (such as ***mov eax, [m]***),

$$\Gamma'(m') = \begin{cases} (k+1, k_w, k_x) & \text{if } m' = m \text{ and} \\ & \Gamma(m) = (k_r, k_w, k_x) \\ \Gamma(m') & \text{otherwise} \end{cases}$$

  A memory address read by an instruction at level $k+1$ has a read level of $k+1$.
- *Memory Write Rule:* if $\nu(\texttt{ip})$ points to an instruction which writes to the memory at address m (such as ***mov [m], eax***),

$$\Gamma'(m') = \begin{cases} (k_r, k+1, k_x) & \text{if } m' = m \text{ and} \\ & \Gamma(m) = (k_r, k_w, k_x) \\ \Gamma(m') & \text{otherwise} \end{cases}$$

  A memory address written by an instruction at level $k+1$ has a write level of $k+1$.

Notice that the execution rule is always applied. Then, depending on the instruction, 0, 1 or both read and write rules need to be applied. For instance, the instruction `xor [edx+esi], eax` both reads from and writes to memory. Indeed, the address pointed by the value of `edx+esi` is read and then written, so if $m = \nu(\texttt{edx}) + \nu(\texttt{esi})$, then $\Gamma'(m) = (k+1, k+1, k_x)$.

Control transfers are evaluated lazily, accounting for indirect, fall-through and asynchronous control transfers. Therefore we require no special semantics for machine instructions other than the memory addresses they read and write.

Now, a well-typed execution is a sequence of typed execution environments $exec(\mu_0, \nu_0, \mathbf{p}) = (\mu_0, \nu_0, \Gamma_0) \to (\mu_1, \nu_1, \Gamma_1) \to \ldots \to (\mu_n, \nu_n, \Gamma_n) \to \ldots$ where Γ_0 is defined by $\Gamma_0(m) = (0, 0, 0)$ for $m \in$ Words.

The level of an execution is defined as the maximal level of an execution environment in a run. Take a well-typed execution environment

$$(\mu_0, \nu_0, \Gamma_0) \to \ldots \to (\mu_n, \nu_n, \Gamma_n) \to \ldots$$

Then, the level of this execution is

$$k = \begin{cases} \max_i\{k_i \mid \Gamma_i \vdash \nu_i(\texttt{ip}) : X(k_i)\} & \text{if defined} \\ \infty & \text{otherwise} \end{cases}$$

A *self-modifying pseudo-program* $\mathbf{p}$ is a pseudo-program $\mathbf{p}$ such that for a given execution environment (μ_0, ν_0), the execution level is strictly greater than 1.

Note that the number of the current layer can both increase and decrease during the execution of the application. The next section introduces changes in the typing system so that the execution level can no longer decrease.

3 Monotonic Execution Waves

3.1 Monotonic typing system

We now introduce a dynamic typing system $\overline{\Gamma}$ for which the execution level is increasing w.r.t. semantics. It will be an advantage to use monotonic typing rules because we then know that we never have to go back in the computation trace in order to determine code layers.

The typing rules of $\overline{\Gamma}$ are identical to the ones of Γ with the exception of the execution rule:

Monotonic Execution Rule:

$$\overline{\Gamma}'(m') = \begin{cases} (k_r, k_w, \max(k_w + 1, k)) & \text{if } m' = \nu(\texttt{ip}) \\ \overline{\Gamma}(m') & \text{otherwise} \end{cases}$$

The read and write rule are identical to Γ ones. Definitions of the previous section are extended in a natural way to $\overline{\Gamma}$.

Suppose that $(\mu_0, \nu_0, \overline{\Gamma}_0) \to (\mu_1, \nu_1, \overline{\Gamma}_1) \to \ldots \to (\mu_n, \nu_n, \overline{\Gamma}_n) \to \ldots$ is a well-typed execution environment. It is not difficult to see that the sequence $(k_i)_i$ of execution levels is increasing where $\overline{\Gamma}_i \vdash \nu_i(\texttt{ip}) : X(k_i)$ for each i.

Take the product ordering on triplets of natural numbers defined as: $(a, b, c) \leq (a', b', c')$ if $a \leq a'$, $b \leq b'$ and $c \leq c'$.

Proposition 1. *For any i and m, we have $\overline{\Gamma}_i(m) \leq \overline{\Gamma}_{i+1}(m)$*

3.2 Code waves

We can study the relation between the executed instruction level by level on one hand, and reading and writing levels on the other hand. For this, we regard a computation as a sequence of code waves, each wave being determined by an execution level. We begin with wave 1 in which the initial code is run. Then, there is wave 2 for which the code has been written by wave 1. The process repeats itself and switches from wave k to $k + 1$ each time that we run code written during wave k.

We consider a well-typed execution environment $(\mu_0, \nu_0, \overline{\Gamma}_0) \to (\mu_1, \nu_1, \overline{\Gamma}_1) \to \ldots \to (\mu_n, \nu_n, \overline{\Gamma}_n) \to \ldots$. Given a step t, we set $X_0^t = \emptyset$ and $X_{k+1}^t = \{m \in \textsc{Words} \mid \overline{\Gamma}_t \vdash m : X(k+1)\}$. So X_k^t is the set of memory locations of execution level k at step t. Then, the set of all memory locations of execution level k in the trace is just $\mathbf{X}_k = \bigcup_t X_k^t$.

We see that all executed instructions during the wave k are pointed by the addresses in $\mathbf{X}_k$. Our goal is now to construct $\mathbf{X}_k$ in order to detect efficiently pseudo-program behaviors and to demonstrate some properties on executions. This leads us to extract an increasing sequence of indices $\ell_1, \ell_2, \ldots, \ell_k, \ldots$ which satisfies:

- the execution level of $(\mu_i, \nu_i, \overline{\Gamma}_i)$ is 1 for each $i \in [0, \ell_1]$,
- the execution level of $(\mu_i, \nu_i, \overline{\Gamma}_i)$ is $k + 1$ for each $i \in [\ell_k + 1, \ell_{k+1}]$,

We now set $\mathbf{SX}(1) = [0, \ell_1]$ and $\mathbf{SX}(k+1) = [\ell_k + 1, \ell_{k+1}]$ the set of indices in which the execution level is 1 (resp. $k+1$). In other words, the set $\mathbf{SX}(k)$ corresponds to the set of time indices of the code wave k.

We can effectively determine $\ell_1, \ell_2, \ldots, \ell_k$ at wave $k+1$ and for any k the sequence $(X^t_{k+1})_t$ has a greatest element, which is the set $X^{\ell_{k+1}}_{k+1}$.

Proposition 2. *For each k, we have $\boldsymbol{X}_{k+1} = X^{\ell_{k+1}}_{k+1}$*

So $\mathbf{X}_{k+1}$, the set of addresses run at wave $k+1$, is computable. Indeed, the above proposition tells us that it is enough to collect all addresses until the execution level increases and becomes $k+1$. Of course, the content of the addresses in $\mathbf{X}_{k+1}$ can change at any time. Therefore, the instructions pointed by addresses at time ℓ_{k+1} may be the wrong ones. However, we can look at the writing level. If $m \in \mathbf{X}_{k+1}$ and $\overline{\Gamma}_{\ell_{k+1}} \vdash m : W(k')$ where $k' \leq k$, then we know that the contents of m is unchanged. Otherwise, the content of m was modified by an instruction of level k. To retrieve the executed instruction, it is then sufficient to find $i \in \mathbf{SX}(k+1)$ such that $\overline{\Gamma}_{\ell_i} \vdash m : W(k')$ where $k' \leq k$. Notice that the existence of i is implied by the typing system. Thus, we are able to retrieve the list of instructions executed at wave $k+1$.

Note also that pseudo-programs deal with new questions on memory location reachability, because a memory address can be reached twice and can point to different instructions. Such addresses belong to $\mathbf{X}_k \cap \mathbf{X}_{k'} \cap (\cup_{k \leq k'' < k'} \mathbf{W}_{k''})$.

The corresponding sets for read and written addresses are then:

- $R^t_0 = \emptyset$ and $R^t_k = \{m \in \textsc{Words} \mid \overline{\Gamma}_t \vdash m : R(k)\}$, so R^t_k is the set of memory locations read during wave k at step t,
- $W^t_0 = \emptyset$ and $W^t_k = \{m \in \textsc{Words} \mid \overline{\Gamma}_t \vdash m : W(k)\}$ so W^t_k is the set of memory locations written during wave k at step t.

Then let $\mathbf{R}_k = \bigcup_t R^t_k$ and $\mathbf{W}_k = \bigcup_t W^t_k$.

Proposition 3. *For each k, we have*

- $\mathbf{R}_{k+1} = R^{\ell_{k+1}}_{k+1}$
- $\mathbf{W}_{k+1} = W^{\ell_{k+1}}_{k+1}$

Intuitively, $\mathbf{R}_{k+1}$ (resp. $\mathbf{W}_{k+1}$) is the set of all memory locations read (resp. written) during the wave $k+1$.

4 Behavior Patterns and Protection Systems

When we run an analysis on a pseudo-program, we collect the typing information, which indicates read, write and execution levels. Moreover the structure of a pseudo-program computation may be seen as divided into waves of executed code. Now, we go further by classifying pseudo-program behaviors. For this, we introduce behavior patterns of pseudo-programs that we illustrate by examples.

A pseudo-program executes code at level k' which was written at level $0 < k < k'$. We construct the set $\mathit{Self}(k, k')$ of locations modified at level k and then executed at level k', as follows

$$\mathit{Self}(k, k') =_{dfn} \mathbf{W}_k \cap \mathbf{X}_{k'} \qquad\qquad 0 < k < k'$$

Then a self-modifying pseudo-program is a pseudo-program such that $\cup_{k<k'} \mathit{Self}(k, k')$ is not empty. The completeness of the definition is ensured by the monotonicity of the typing system. The soundness and completeness are stated by:

Proposition 4. *Let* $(\mu_0, \nu_0, \overline{\Gamma}_0) \rightarrow (\mu_1, \nu_1, \overline{\Gamma}_1) \rightarrow \ldots \rightarrow (\mu_n, \nu_n, \overline{\Gamma}_n) \rightarrow \ldots$ *be a well-typed execution environment. For any* k *and* k' *such that* $0 < k < k'$, $m \in \mathit{Self}(k, k')$ *iff* $\exists t_1, t_2$ *such that* $t_1 \leq t_2$, $\overline{\Gamma}_{t_1}(m) = (k_r, k, k_x)$ *and* $\overline{\Gamma}_{t_2}(m) = (k_r, k, k')$.

We can now define some usual behavior patterns as first-order formulas over the predicates $\mathbf{R}_k$, $\mathbf{W}_k$, $\mathbf{X}_k$.

- *Blind Self-Modification*: Wave k performs a blind self modification on wave k' if

$$\mathit{Blind}(k, k') =_{dfn} \mathit{Self}(k, k') \setminus \mathbf{R}_k \neq \emptyset$$

- *Decryption*: Wave k decrypts wave k' if

$$\mathit{Decrypt}(k, k') =_{dfn} \mathit{Self}(k, k') \cap \mathbf{R}_k \neq \emptyset$$

- *Integrity Checking*: Wave k checks the integrity of wave k' if

$$\mathit{Check}(k, k') =_{dfn} \mathbf{R}_k \cap \mathbf{X}_{k'} \setminus \cup_{min(k,k') \leq k'' \leq max(k,k')} \mathbf{W}_{k''} \neq \emptyset$$

- *Code Scrambling*: Wave k is scrambled by wave k' if for $k < k'$

$$\mathit{Scrambled}(k, k') =_{dfn} \mathbf{X}_k \cap \mathbf{W}_{k'} \neq \emptyset$$

Of course, we can define other behavior patterns. In all cases, a pseudo-program satisfies a behavior pattern A, $A \in \{\mathit{Blind}, \mathit{Decrypt}, \mathit{Check}, \mathit{Scrambled}\}$, if $\cup_{k,k'} A(k, k')$ is not empty.

We see that a lot of pseudo-program behaviors can be described as boolean combinations of base predicates $\mathbf{R}_k$, $\mathbf{W}_k$, $\mathbf{X}_k$. So, we can detect a particular behavior by establishing that a behavior pattern is true (i.e. not empty). For example, we can determine whether or not a pseudo-program has a blind self-modifying behavior or decrypts some code. The key point here is that a such pattern analysis may raise some warnings when a suspicious behavior is recognized. For example, we might raise a warning when a pseudo-program modifies it own code and executes it like in section 1.5. Practical issues related to pattern detection are presented in subsection 5.1.

4.1 Mandatory RWX-controls

We now consider the issue of verifying non-interference properties. Information-flow properties allow to enforce the confidentiality and the integrity of data [29]. These models are designed for programming languages with a fixed program text and so focus on static analysis. When the program source is not available, information-flow properties can only be enforced at run time [7,3]. Although the situation is different in our setting, we establish some information-flow characterizations related to non-interference properties. Intuitively, we can determine whether a memory location is needed in the computation of a wave. The first Lemma is a confinement property, it says that in wave $k+1$, only locations in $\mathbf{W}_{k+1}$ have their contents modified. Therefore, all other locations (i.e. not in $\mathbf{W}_{k+1}$), are unchanged.

Lemma 1. *[Confinement] For any $i, i' \in \boldsymbol{SX}(k+1)$ and any address $m \notin \boldsymbol{W}_{k+1}$, we have $\mu_i(m) = \mu_{i'}(m)$.*

Proof. Suppose for the sake of contradiction that $\mu_i(m) \neq \mu_{i'}(m)$ and $i < i'$. In this case, an instruction at step j writes to m where $i < j \leq i'$. So $\overline{\Gamma}_{\ell_j} \vdash m : \mathbf{W}_{k+1}$ and therefore $m \in \mathbf{W}_{k+1}$.

We now establish the soundness of the $\overline{\Gamma}$ type system with respect to the semantics. The soundness theorem states that if $m \in \mathbf{R}_{k+1} \cup \mathbf{W}_{k+1} \cup \mathbf{X}_{k+1}$, then we can arbitrarily alter the value of location m' in wave $k+1$ such that $m' \notin \mathbf{R}_{k+1} \cup \mathbf{X}_{k+1}$, execute the pseudo-program steps in wave $k+1$, and the value of m will be the same.

Theorem 1. *[Type soundness] Assume that $\ldots (\mu_\ell, \nu_\ell, \overline{\Gamma}_\ell) \to \ldots \to (\mu_{\ell+t}, \nu_{\ell+t}, \overline{\Gamma}_{\ell+t})$ is a well-typed execution environment. Assume also that ℓ and $\ell + t$ are in $\boldsymbol{SX}(k+1)$. Suppose that*

(a) $(\mu'_\ell, \nu_\ell, \overline{\Gamma}_\ell) \to \ldots \to (\mu'_{\ell+t}, \nu'_{\ell+t}, \overline{\Gamma}'_{\ell+t})$ is another execution environment,
(b) $\mu_\ell(m) = \mu'_\ell(m)$ for all $m \in \mathbf{R}_{k+1} \cup \mathbf{W}_{k+1} \cup \mathbf{X}_{k+1}$.

Then,

1. $\overline{\Gamma}_{\ell+t}(m) = \overline{\Gamma}'_{\ell+t}(m)$ for all addresses m
2. $\nu_{\ell+t}(rg) = \nu'_{\ell+t}(rg)$ for all registers rg
3. $\mu_{\ell+t}(m) = \mu'_{\ell+t}(m)$ for all $m \in \mathbf{R}_{k+1} \cup \mathbf{W}_{k+1} \cup \mathbf{X}_{k+1}$.

Proof. The proof goes by induction on the number of steps t.

Suppose that the induction hypothesis is satisfied until step t. First, we show that both runs execute the same instruction. By induction hypothesis on (2), we have $\nu_{\ell+t}(\texttt{ip}) = \nu'_{\ell+t}(\texttt{ip})$. Since $\nu_{\ell+t}(\texttt{ip}) \in \mathbf{X}_{k+1}$, we have $\mu_{\ell+t}(\nu_{\ell+t}(\texttt{ip})) = \mu'_{\ell+t}(\nu'_{\ell+t}(\texttt{ip}))$ by induction hypothesis on (3).

– *Execution Rule:* Induction hypothesis (1) and (2) imply that $\overline{\Gamma}_{\ell+t+1}(\nu_{\ell+t}(\texttt{ip})) = \overline{\Gamma}'_{\ell+t+1}(\nu'_{\ell+t}(\texttt{ip}))$.

– *Memory Write Rule:* $\nu(\texttt{ip})$ points to an instruction which writes to the memory at address m (such as *`mov [m], eax`*). There are three cases:

  1. By induction hypothesis, $\overline{\Gamma}_{\ell+t}(m) = \overline{\Gamma}'_{\ell+t}(m) = (k_r, k_w, k_x)$. After executing the instruction and following the $\overline{\Gamma}$ typing rule, we obtain that $\overline{\Gamma}_{\ell+t+1}(m) = \overline{\Gamma}'_{\ell+t+1}(m) = (k_r, k+1, k_x)$. This shows that (1) holds.
  2. Registers are unchanged, so (2) holds.
  3. By induction hypothesis, we have $\nu_{\ell+t}(\texttt{eax}) = \nu'_{\ell+t}(\texttt{eax})$, so $\mu_{\ell+t+1}(m) = \mu'_{\ell+t+1}(m)$. And (3) holds.

– *Memory Read Rule:* It is similar to the previous case.

5 Applications

5.1 Malware analysis

Behavioral signatures and packer visualization We can visualize the behavior of pseudo-programs as a directed graph:

– nodes are code waves
– edges are the relations between the waves. For instance, there will be an edge between waves k and k' labeled "decrypts" if $\mathit{Decrypt}(k, k')$ is not empty.

As far as we know, this visualization is new and can give a quick understanding of the behavior of complex packers. As an example, figure 2 is the self-reference graph of Yoda Protector.

These graphs could potentially be used as signatures. The idea of using behavioral graphs as signatures is not new [36], but to the best of our knowledge extending the notion of behavior with self-reference and self-modifications has not been published before

Experimentation In [15], we developed a prototype implementation based on this framework. This implementation was then run on a large number of malware samples and distributed on a cluster. Here is a highlight of the results:

– approximately 60,000 samples (with unique md5 hashes) have been collected on a Nepenthes honeypot
– each binary was executed in its own Windows virtual machine for 2 minutes
– these binaries have been analyzed in 35 hours on a 12 nodes cluster
– 81.28% of these binaries were analysed successfully
– although PEiD detected a packer in only 2.92% of the samples, we detected self-modifying code in 80.74%

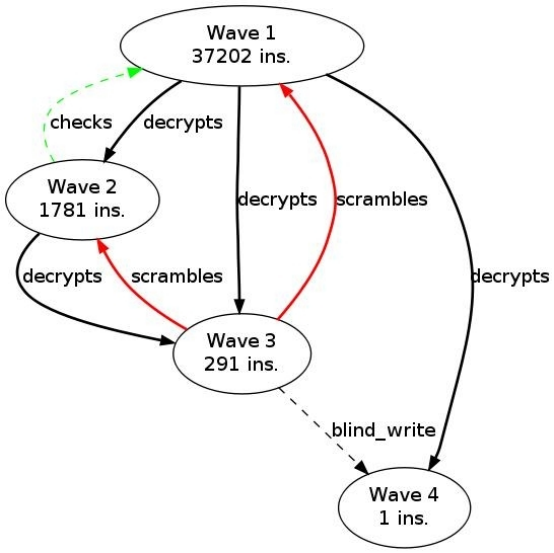


Fig. 2. Yoda Protector Self-Reference Graph

5.2 Other potential uses

The notion of pseudo-programs and the formalism around it is so general that it can be used in different areas of program analysis:

- *behavioral monitoring*: as explained in 4, a security policy can be defined as a first order formula over the sets $\mathbf{R}_k$, $\mathbf{W}_k$ and $\mathbf{X}_k$. We can then enforce this policy on untrusted binaries and prevent dangerous behaviors such as nested self-modifications. To limit the performance cost, a possibility is to use a lightweight instrumentation framework such as VX32 [13].
- *fuzzing*: the non-interference theorems (section 4.1) give the memory locations that have an impact on a given wave. It allows the fuzzer to target directly the inputs of this particular wave.
- *vulnerability research*: a key observation is that a class of memory corruption vulnerabilities (such as buffer overflows) can be seen as unintentional self-modifying code. Indeed, the goal of the attacker is to send a particular input to the vulnerable program, in order to disrupt the original control flow and execute the shellcode. As a consequence, this class of vulnerabilities can be detected and analysed with the dynamic typing rules described earlier.

6 Limitations

In the method we describe, the *data-flow in the program is inferred* rather than actually followed. For example, if a given memory address is first read, then written and finally executed, we assume that there is some data-flow between

the read data and the written data. This assumption can be wrong in some cases, and lead to wrong labeling (decrypted code instead of blind self-modification). Another case of bad labeling can occur if the decrypted data is stored at a different memory location than the encrypted data. In this case, no data-flow is inferred and the code is labeled blind self-modification instead of decrypted code.

Unlike [31], we do not address the problem of *embedded interpreters* (or emulators). Since we monitor the native memory for intersections between executed, read and written addresses, the method can not be applied if all the code protection happens in a virtual memory model. We will still see the reads and writes to native memory, but with no knowledge of the interpreter we have no access to the virtual instruction pointer and therefore we can not follow the virtual instruction flow. Since all the patterns used in code labeling involve the execution of a given memory address, no code labeling can occur in interpreted code.

Our approach is mostly dynamic so we face the *single-path execution problem*. However, a multiple path exploration algorithm [25] is not incompatible with the technique we describe.

Finally, we did not address the problem of *multiprocessor programs* [30] in this paper, but the pseudo-programs semantics can be extended to multiprocessor pseudo-programs.

Conclusion

We defined semantics for pseudo-programs, i.e. programs with listings evolving over time. Our formalism allows us to extract a structure from this temporal evolution and to define a taxonomy of self-modifying programs. Code waves can be reconstructed precisely given a fine-grained run trace and we are able to detect the use of common code armoring techniques by computing first-order predicates on these code waves.

From a theoretical perspective, this study, and in particular the dynamic typing system, is a starting point to have a better vision of self-modifying programs. We are investigating the use of this dynamic typing system to deploy untrusted binaries safely on embedded systems. The non-interference properties also raise interesting questions. As far as we know, those non-interference results are new in this context. There are at least two challenging directions to explore: (i) defining code protection by dynamically tiered memory and (ii) vulnerability research.

From a practical perspective, an interesting issue is the code instrumentation. We are thinking of lightweight binary instrumentation, in the spirit of [14] and [13], in order to enforce security policies on processes.

References

1. Bochs: The open source ia-32 emulation project. http://bochs.sourceforge.net/.
2. The ida pro disassembler and debugger. http://www.hex-rays.com/idapro/.

3. M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity: Principles, implementations, and applications. *To appear in ACM Transactions on Information and System Security*.
4. Lutz Boehne. Pandora's bochs: Automatic unpacking of malware. 2008.
5. Guillaume Bonfante, Jean-Yves Marion, and Daniel Reynaud. A computability perspective on self-modifying programs. In *7th IEEE International Conference on Software Engineering and Formal Methods - SEFM 2009*, Hanoi Viet Nam, 11 2009.
6. H. Cai, Z. Shao, and A. Vaynberg. Certified self-modifying code. *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'07)*, 2007:66–77, 2007.
7. A. Chaudhuri, P. Naldurg, and S. Rajamani. A type system for data-flow integrity on windows vista. In *ACM Workshop on Programming Languages and Analysis for Security (PLAS)*, 2008.
8. DataRescue. Using the universal pe unpacker plug-in included in ida pro 4.9 to unpack compressed executables, 2005.
9. S. Debray, K. Coogan, and G. Townsend. On the semantics of self-unpacking malware code. 2008.
10. Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. Ether: malware analysis via hardware virtualization extensions. In *CCS '08: Proceedings of the 15th ACM conference on Computer and communications security*, pages 51–62, New York, NY, USA, 2008. ACM.
11. Gergely Erdélyi. Idapython: User scripting for a complex application, 2008. Bachelor's Thesis, EVTEK University of Applied Sciences.
12. Peter Ferrie. Anti-unpacker tricks. 2008.
13. Bryan Ford and Russ Cox. Vx32: lightweight user-level sandboxing on the x86. In *ATC'08: USENIX 2008 Annual Technical Conference on Annual Technical Conference*, pages 293–306, Berkeley, CA, USA, 2008. USENIX Association.
14. Vanessa Gratzer and David Naccache. Alien vs. quine. *IEEE Security and Privacy*, 5(2):26–31, 2007.
15. Wadie Guizani, Jean-Yves Marion, and Daniel Reynaud. Server-side dynamic code analysis. In *4th International Conference on Malicious and Unwanted Software*, 2009.
16. Carl A. Gunter. *Semantics of Programming Languages: Structures and Techniques*. MIT Press, 1992.
17. Fanglu Guo, Peter Ferrie, and Tzi-Cker Chiueh. A study of the packer problem and its solutions. In *RAID '08: Proceedings of the 11th international symposium on Recent Advances in Intrusion Detection*, pages 98–115, Berlin, Heidelberg, 2008. Springer-Verlag.
18. Sebastien Josse. Secure and advanced unpacking using computer emulation. *Journal in Computer Virology*, 3, 2007.
19. Min Gyung Kang, Heng Yin, and Pongsin Poosankam. Renovo: A hidden code extractor for packed executables. In *5th ACM Workshop on Recurring Malcode*, 2007.
20. S.C. Kleene. *Introduction to Metamathematics*. Van Nostrand, 1952.
21. Mario Alberto Lopez. Unpacking, a hybrid approach. In *2nd International CARO Workshop*, 2008.
22. Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Kim Hazelwood, and Vijay Janapa Reddi. Pin: Building customized program analysis tools with dynamic instrumentation. In *Programming Language Design and Implementation (PLDI)*, 2005.

23. L. Martignoni, M. Christodorescu, and S. Jha. Omniunpack: Fast, generic, and safe unpacking of malware. In *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*, pages 431–441, Dec. 2007.
24. J. Mitchell. *Foundations for Programming Languages*. MIT Press, 1996.
25. Andreas Moser, Christopher Kruegel, and Engin Kirda. Exploring multiple execution paths for malware analysis. In *SP '07: Proceedings of the 2007 IEEE Symposium on Security and Privacy*, pages 231–245, Washington, DC, USA, 2007. IEEE Computer Society.
26. Magnus O. Myreen. Verified just-in-time compiler on x86. In *Proceedings of Principles of Programming Languages (POPL)*, 2010. incomplete reference.
27. Danny Quist and Valsmith. covert debugging, circumventing software armoring techniques. In *Black Hat USA*, 2007.
28. Paul Royal, Mitch Halpin, David Dagon, Robert Edmonds, and Wenke Lee. Polyunpack: Automating the hidden-code extraction of unpack-executing malware. In *ACSAC '06: Proceedings of the 22nd Annual Computer Security Applications Conference on Annual Computer Security Applications Conference*, pages 289–300, Washington, DC, USA, 2006. IEEE Computer Society.
29. A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, January 2003.
30. Susmit Sarkar, Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Tom Ridge, Thomas Braibant, Magnus O. Myreen, and Jade Alglave. The semantics of x86-cc multiprocessor machine code. In *POPL '09: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 379–391, New York, NY, USA, 2009. ACM.
31. Monirul Sharif, Andrea Lanzi, Jonathon Giffin, and Wenke Lee. Automatic reverse engineering of malware emulators. In *SP '09: Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*, pages 94–109, Washington, DC, USA, 2009. IEEE Computer Society.
32. Skape. Using dual-mappings to evade automated unpackers. 2008.
33. Joe Stewart. Semi-automatic unpacking on ia-32. In *Defcon*, 2006.
34. Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Cross site scripting prevention with dynamic data tainting and static analysis. February 2007.
35. G. Winskel. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, 1993.
36. Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. Panorama: capturing system-wide information flow for malware detection and analysis. In *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security*, pages 116–127, New York, NY, USA, 2007. ACM.

A Certified JavaScript Interpreter

Martin Bodin[1] & Alan Schmitt[2]

1: Projet Celtique, Inria Rennes – Bretagne-Atlantique et ENS Lyon
martin.bodin@inria.fr
2: Projet Celtique, Inria Rennes – Bretagne-Atlantique
alan.schmitt@inria.fr

Introduction

The JavaScript language was initially developed for web pages enrichment, allowing the execution of scripts by the browser. It is now pervasively used on the web, not only to add interactivity in websites or to embed contents from third-party sources, but also as a target platform for deploying applications written in other languages (such as ocaml bytecode [VB11], Hop [SGL06], or LLVM assembly [Zak11]). In some sense, JavaScript has become the assembly language of the web, as most browsers are now able to run it. More recently, it has been used to program user interfaces for embedded systems, such as the defunct WebOS (now Enyo [Eny12]), the Kindle Touch ebook reader, or for the BootToGecko project [Moz12].

In addition to its pervasive use, JavaScript presents two important characteristics. First, as it was initially developed to facilitate its integration with the browser and with web contents, it aims more at providing powerful features than at giving robustness and safety guarantees. These powerful features include first class functions and closures, prototype-based objects, dynamic typing with many conversion functions, explicit scope manipulation, and the evaluation of strings as code. A second, redeeming, characteristic of JavaScript is that it is standardized [A+99], providing more information about how these features interact.

The goal of the JSCert project [BCF+12] is to provide a precise and formal semantics to JavaScript to build tools to certify analyses and compilation procedures. JSCert's collaborators have defined such a semantics in the Coq proof assistant, based both on the paper formalization of Maffeis et al. [MMT11,MMT08] and on the specification. To gain and provide more confidence in this formalization, we have implemented an interpreter that is proven correct in relation to the semantics. We will thus be able to confront our semantics against JavaScript test suites.

This paper describes the design and implementation of the interpreter. It is organized as follows. Section 1 introduces the semantics of JavaScript and highlights some of its peculiarities. Section 2 describes the interpreter's design and implementation. Section 3 addresses the interpreter's correctness. Finally, Section 4 concludes with future and related work.

1. JavaScript 's Semantics

JavaScript is defined by the standards ECMAScript 3 and 5 (ES3 and ES5) [A+99]. Most web browsers implement every feature of ES3 as well as some of ES5. Some also provide features that are not standardized, such as the modification of the implicit prototype of an object. The formalization described here is based on ES3 without any unspecified extension.

1.1. Memory Model

The execution context of a JavaScript program comprises two objects: a *heap* and a *scope chain*. Objects in the heap are indexed by *locations*. Objects are maps from fields to values (including locations). Intuitively, locations can be seen as pointers.

The scope chain is a stack of locations (called *scopes* when they are in this stack). The top of the stack is a special location l_g pointing to the *global object*, where every global variable of the running program resides. When looking up the value of a variable `x`, it is searched in the scope chain. More precisely, the value of `x` will be found in the first location in the scope chain where it is defined. This behavior is similar to the one of a lexical scope (local variables having priority over global variables of the same name). However, as scopes are usual objects, they can be dynamically modified. Moreover, we will see below that scopes can be manually added to the chain using the `with` operator.

Variable look-up is also determined by the *prototypes* of the objects under consideration. Every object has an implicit prototype (in the form of a special field that *should not* be accessible, which we call @*proto*), possibly pointing to the special location `null`. Unless the prototype is `null`, it also has an implicit prototype, and so on, forming a prototype chain. The semantics of JavaScript guarantees that no loop can appear in the prototype chain. Intuitively, the field @*proto* of a location l points to a location representing the class from which l inherits. More precisely, each time a field `x` of a location l is looked up, l is checked to effectively have this field. If it is not the case, the prototype chain is followed until such an `x` is found.

We now describe how this mechanism interacts with the scope chain. Figure 1 shows an example of a JavaScript scope chain, where horizontal arrows depict the prototype chains. To access variable `x` in the current scope l_0, it is first searched in l_0 itself and its prototype chain. As `x` is not found, the scope chain is followed and the variable is looked up in l_1 and its prototype chain. This time, `x` is found in location l_2, thus the value returned is 1. Note that the value 2 of `x` present in l_g is shadowed by more local bindings, as well as the value 3 present in l_3.

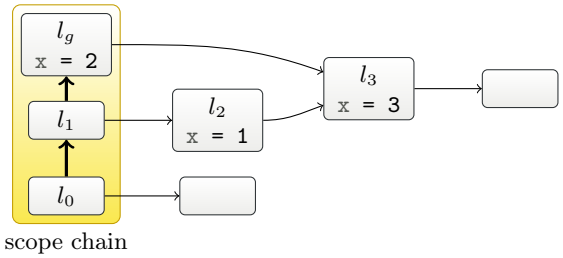
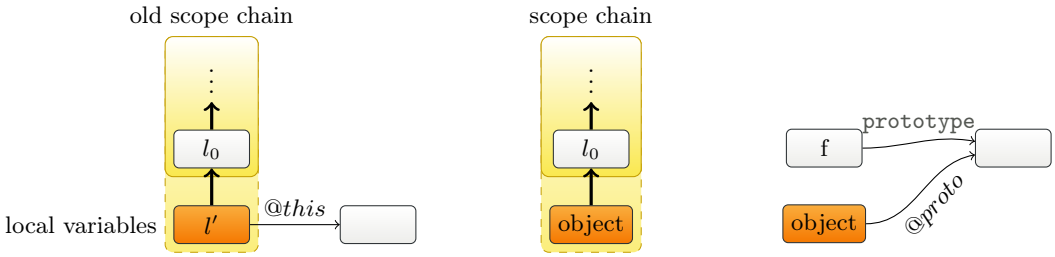


Fig. 1: A JavaScript scope chain

Some special objects have a particular use. We have already encountered the global object, located at l_g. This object is where global variables are stored. As an object, its field @*proto* is bound to l_{op}, which we describe below. The global object is always at the top of the scope chain. A second special object is the prototype of all objects, `Object.prototype`, located at l_{op}. Every newly created object has a field @*proto* that is bound to l_{op}. It has some functions that thus can be called on every object (but they can be hidden by local declarations) such as `toString` or `valueOf`. Finally, the prototype l_{fp} of all functions, `Function.prototype`, is a special object equipped with function-specific methods.

1.2. Manipulating Heaps

The model presented above shows how a read is performed in a heap. Let us now see how the scope chain is changed over the execution of a program, typically because of the execution of a function call, a `with` statement, a `new` statement, and an assignment. A graphical representation of those changes is summed up in Figure 2. In this figure, the "▭" orange blocks represents newly allocated locations.



(a) Function Call. (b) `with`. (c) `new`.

Fig. 2: Scopes chain manipulation

As usual in functional programming language, the current scope chain is saved when defining new functions. Upon calling a function, the scope chain is restored, adding a new scope at the front of the chain to hold local variables and the arguments of the call. A special field @*this*, used by the `new` rule, is also added. The `with(o){...}` statement puts object `o` in front current scope chain to run the associated block. In the `new f (...)` case, function `f` is called with the special field @*this* assigned to a new object. The implicit prototype @*proto* of this new object is bound to the object pointed by the `prototype` field of `f`. This is how immutable prototype chains are created. The newly created object, which may have been modified during the call to `f` by "`this.x = ...`" statements, is then returned.

Example. A heap modified by a `new` operator called at Line 7 of the program of Figure 3a is shown in Figure 3. As some locations (such as `null`, l_{op}, or l_{fp}) are often used as implicit prototypes, they are put at the bottom left of locations instead of being explicitly depicted in the graph. Upon executing the `new` instruction, function `f` is called, adding a new location (the dashed one in Figure 3c) for the function call. In this location, the argument `a` is set to 42, and the @*this* field points to a new object (red in the figure). The function body is executed, which adds an `x` field to the red object. This object is then returned, setting its implicit prototype @*proto* to the value of the field `prototype` of `f`.

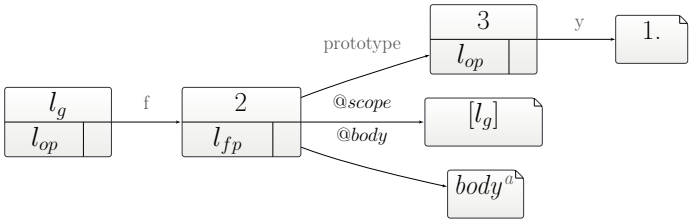
Targeted assignment, of the form `l.x = 3`, are straightforward: variable `x` is written with value `3` in the object at location `l`. For untargeted assignments, such as `x = 3`, things are more complex. The first scope for which the searched field is defined is selected in the current scope chain, following the same variable look-up rules as above. The variable is then written in the found scope. If no such scope is found, then a new variable is created in the global scope.

Figure 4 describes the assignment `x = 3`. Location l_1 is the first to define `x` in its prototype chain (in l_2). The new value of `x` is then written in l_1. Note that it is not written in l_2, allowing other objects that have l_2 in their prototype chain to retain their old value for `x`. Nevertheless, if one accesses `x` in the current scope chain, the new value 3 is returned. This allows objects constructed from the same function `f` (using `new`)—which thus have the same implicit prototype—to share some values, while letting them set those values without changing the shared one, similar to a *copy-on-write* approach. This approach may lead to surprising behaviors, as we now illustrate.

Example. Let us consider the following program.

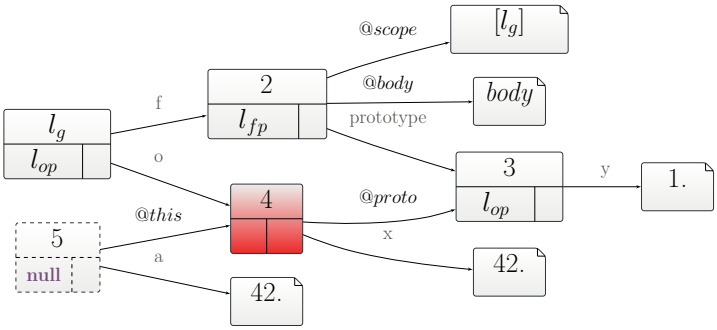
```
1  var f = function(a) {
2    this.x = a;
3  }
4
5  f.prototype = {y :
       1};
6
7  var o = new f (42);
```

(a) The executed program.



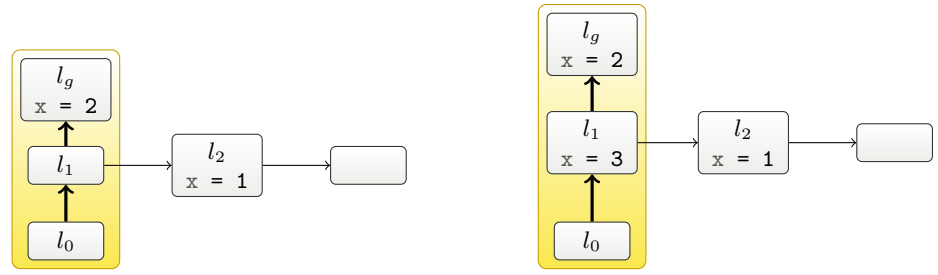
(b) Heap's state at Line 6.

[a] This is the body of the function `f`, that is `function(a){ this.x = a ; }`.



(c) Heap's state at the end of Line 7.

Fig. 3: Effect of the `new` instruction



(a) Before.

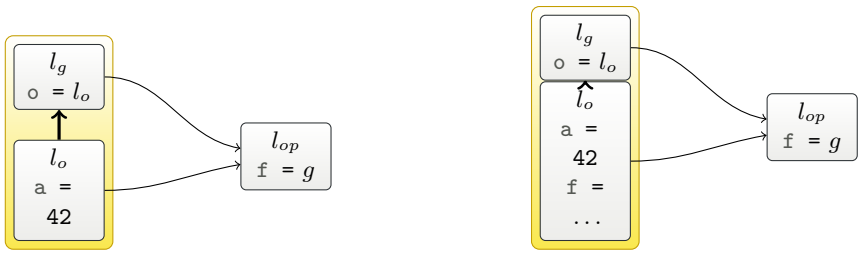
(b) After.

Fig. 4: Assignment

```
1 var o = {a : 42};
2 with (o) {
3   f = function() {return a;};
4 };
5 f ()
```

If it is executed in an empty heap, it returns `42`. Indeed, when defining `f`, no such function already exists, thus `f` is stored in the global scope. When `f` accesses `a` upon its call, the object `o` is in the scope chain (as the call is executed in the scope chain of the definition of `f`), thus the result is `42`.

Let us now consider it in a slightly different scope, where `Object.prototype.f` has been set to a given function (say g =`function(){ return 18; }`). As the code `var o = {a : 42}` is almost equivalent to `var o = new Object(); o.a = 42`, object `o` has an implicit prototype set to l_{op}, which is `Object.prototype`. Figure 5 shows a representation of the heap at Line 3. As there is a variable `f` defined in the scope chain at position l_o (because of its prototype chain to l_{op}), the assignment is local to l_o and not global. At Line 5, the variable look-up for `f` returns the function g which is found in the prototype of the global object l_g (at this point the scope chain only contains l_g), and not the f defined in the `with` block. Thus the call returns `18`.



(a) at the beginning of Line 3.

(b) at the end of Line 3.

Fig. 5: Heap state of the program in Page

There are many other subtleties in JavaScript's semantics which can be used to execute arbitrary code. For instance, implicit type conversion uses `toPrimitive` conversion functions, which could be redefined in a way similarly difficult to detect.

2. The Interpreter

2.1. Overview

We now describe the main contribution of this paper: an interpreter that rigorously follows a subset of the ES3 standard, more precisely a Coq-formalization of it. This formalization is based on the big step semantics presented in [GMS12] and covers a significant part of the core language, rich enough to observe some complex behaviors. We also rely on a formalization of a small-step-semantics [MMT11,MMT08], in particular for the primitive operators not formalized in the big-step-semantics.

The Coq formalization follows the specification closely and makes the distinction between expressions, statements, and programs. It includes function declarations, tests, `while` loops, `with` and `try`...`catch`...`finally` blocks, `throw` instructions, function calls, `new`, `delete`, assignments, construction of objects, objects accessors, `this`, `typeof`, and most unary and binary operators (including boolean

lazy operators). We are currently working on adding type conversion and `eval` to this semantics. Labels and `break` statements are not yet supported.

This interpreter, proven correct with respect to our semantics, will allow to confront our semantics against JavaScript test suites. This will highly enforce its trustability.

The trusted base of the Coq formalization is composed of the three files `JsSyntax.v`, `JsSemantic.v`, and `JsWf.v` that respectively define the syntax, the semantics, and the invariants over heaps. They are fairly short (600 loc) when compared to the interpreter and the associated proofs (1800 loc). We will give more details about the correctness proof in Section 3.2.

The interpreter takes as arguments the initial heap, a scope chain, a program to be executed, and a bound on the number of reduction steps. This last argument is mandated by the termination requirement of Coq. The interpreter returns either `out_bottom`, when the bound is not large enough, for instance when the input program loops, or `out_return` when a result or an exception is returned.

2.2. Dependencies

The development relies heavily on the TLC library [Cha10b]. This library includes very powerful automation tactics, which has been especially useful to maintain the proofs as additional JavaScript constructs are added. We also use the Flocq library [BM10] to model `IEEE` floats in Coq.

In the spirit of TLC, proofs and definitions have mostly been done in a classical setting, which simplifies the development but requires additional care during extraction. For instance, consider the TLC expression `If x = y then e1 else e2`. It is defined as `if classicT (x = y)then v1 else v2`, where `classicT` is a lemma of type $\forall$(`P : Prop`), {`P`} + {¬`P`}. Such an expression is extracted to OCaml to `if isTrue then e1' else e2'` where `isTrue` is defined by an exception, independently of `x` and `y`.

```
(** val isTrue : bool **)

let isTrue =
  if let h = failwith "AXIOM TO BE REALIZED" in if h then true else false
  then true
  else false
```

To address this issue, decidable tests are associated to classical ones in a fairly transparent way, by defining for each test a boolean function `decide` to be used during extraction. This approach can be considered as a form of small-scale reflection, packing together a proposition stating a property and a decidable boolean function computing it is truth value.

```
Class Decidable (P:Prop) := make_Decidable {
  decide : bool;
  decide_spec : decide = isTrue P }.
```

The use of type classes [SO08] lets us avoid mentioning which function `decide` shall be used. We can now replace the code `If x = y then e1 else e2` by `if decide (x = y)then e1 else e2` (which we abbreviate to `ifb x = y then e1 else e2` for short). During the OCaml extraction, the comparison `if field_comparable x y then e1 else e2` is generated.

We also rely on an *optimal fixed point* to define the function π of [GMS12]. Function π searches through a prototype chain for a given variable and returns the location where it has been found (or `null` if not). It is defined formally in Figure 6, where $\mathrm{dom}(H)$ represents all the pairs of locations and field defined in the heap H, and $H(l, x)$ is the associated value for (l, x).

To facilitate the extraction of such a function, we use the optimal fixed point library [Cha10a]. This lets us separate the definition of the function from the proof that it terminates, avoiding in particular the use of dependent types in the definition. More precisely, we first define a function that performs one step of reduction. Here is how it is defined for the π function.

$$\frac{}{\pi(H, \texttt{null}, x) \triangleq \texttt{null}} \qquad \frac{(l,x) \in \mathrm{dom}(H)}{\pi(H,l,x) \triangleq l} \qquad \frac{(l,x) \notin \mathrm{dom}(H) \quad H(l, @proto) = l'}{\pi(H,l,x) \triangleq \pi(H,l',x)}$$

Fig. 6: Reduction rules associated with the function π.

```
1  Definition proto_comp_body proto_comp h f l :=
2    ifb l = loc_null then loc_null
3    else ifb indom h l f then l
4    else ifb indom h l field_proto then
5       match read h l field_proto with
6       | val_loc l' ⇒proto_comp h f l'
7       | _ ⇒arbitrary
8       end
9     else arbitrary.
```

The identifier `arbitrary` at Lines 7 and 9 represents an arbitrary element of the return type (here a location), which is available as the return type is proven to be inhabited. In the extracted code, those calls to `arbitrary` are replaced by exceptions and should never happens (Section 3.1 details the invariants of the heap that guarantee this). The final value for `proto_comp` is defined using the operator `FixFun3`, which will be extracted to OCaml as a `let rec` construction.

```
1  Definition proto_comp := FixFun3 proto_comp_body.
```

No proof is performed there as `FixFun3` is defined using (in addition with some classical logic) the predicate `Fix_prop` defining the greatest fixed point of `proto_comp_body`.

```
1  Fix_prop = fun (A : Type) (E C : binary A) (F : A → A) (x : A) ⇒
2     greatest C (fixed_point E F) x
```

These steps are sufficient to define a function that can be extracted. Note that no property has been shown about this function. To be able to reason about it, we need to show that it is actually a fixpoint and that recursive calls are made according to a decreasing measure.

```
1  Lemma proto_comp_fix : ∀h f l,
2    ok_heap h →proto_comp h f l = proto_comp_body proto_comp h f l.
```

In the proof of the preceding lemma, the only way to destruct the `FixFun3` operator is by a lemma that requires a decreasing measure related to `proto_comp_body` arguments. We can then use this fixpoint relation when reasoning about `proto_comp`. See [Cha10a] for additional details on optimal fixed points.

2.3. Structure of the Interpreter

The semantics and the interpreter are based on a presentation derived from the big-step-semantics, called *pretty-big-step-semantics*, described in [Cha12]. This presentation allows to factorize error and exception handling, as in a small-step presentation, while writing big-step rules.

Let us take the example of the rule associated to variable assignment in a big-step-semantics. This rule takes an initial heap H, a scope chain L, and an expression `e` to a final heap H_3 and a value `v`. We write $H[\texttt{l} \leftarrow \texttt{v}]$ for the heap H where location[1] `l` has been updated with value `v`.

$$\frac{H, L, \texttt{e1} \longrightarrow H_1, \texttt{Ref l} \quad H_1, L, \texttt{e2} \longrightarrow H_2, \texttt{v} \quad H_3 = H_2[\texttt{l} \leftarrow \texttt{v}]}{H, L, \texttt{e1} = \texttt{e2} \longrightarrow H_3, \texttt{v}}$$

[1] Technically `l` is a pair of a location and a field name, but to simplify the presentation we leave it abstract.

This rule is simple, but it does not take into account errors and exceptions. The problem is that adding those constraints would lead to some duplications in this rule: the part $H, L, \texttt{e1} \longrightarrow \ldots$ executing `e1` would appear four times:

$$\frac{H, L, \texttt{e1} \longrightarrow H_1, \texttt{exn}}{H, L, \texttt{e1} = \texttt{e2} \longrightarrow H_1, \texttt{exn}}$$

$$\frac{H, L, \texttt{e1} \longrightarrow H_1, \texttt{v} \quad \texttt{v} \neq \texttt{Ref l}}{H, L, \texttt{e1} = \texttt{e2} \longrightarrow H_1, \texttt{exn}}$$

$$\frac{H, L, \texttt{e1} \longrightarrow H_1, \texttt{Ref l} \quad H_1, L, \texttt{e2} \longrightarrow H_2, \texttt{exn}}{H, L, \texttt{e1} = \texttt{e2} \longrightarrow H_2, \texttt{exn}}$$

$$\frac{H, L, \texttt{e1} \longrightarrow H_1, \texttt{Ref l} \quad H_1, L, \texttt{e2} \longrightarrow H_2, \texttt{v} \quad H_3 = H_2[\texttt{l} \leftarrow \texttt{v}]}{H, L, \texttt{e1} = \texttt{e2} \longrightarrow H_3, \texttt{v}}$$

Note that only exceptions are dealt with there: the same duplication occurs for all the instructions breaking the normal control flow (such as `return` or `break`). This uselessly duplicates some rules, making them difficult to read and increasing the redundancy in the proofs.

The pretty-big-step semantics consists in splitting the assignment rule to several sub-rules representing the partial reductions of the arguments of the assignment, as if we were defining a small-step-semantics. To this end, outcomes o that are pairs of the final heap and either a value or an exception are added. The expressions $o =_1 \texttt{e}$ and $\texttt{l} =_2 o$ are also introduced, as well as a new predicate **abort** o. This predicate is satisfied when o is $(H, \texttt{exn})$. The rule for $\texttt{e1} = \texttt{e2}$ is now split between computing (and eventually propagation of special results) of its sub-expressions `e1` and `e2`, and in the effective computing of the assignment. Note that in rules of the form $H : o_1 =_1 \texttt{e2} \Downarrow o$, the heap H is not taken into account, it is the heap in o_1 that will be used for the remainder of the computation. Figure 7 shows those new rules.

$$\frac{H : \texttt{e1} \Downarrow o_1 \qquad H : o_1 =_1 \texttt{e2} \Downarrow o}{H : \texttt{e1} = \texttt{e2} \Downarrow o} \qquad \frac{\textbf{abort}\, o}{H : o =_1 \texttt{e2} \Downarrow o} \qquad \frac{v \neq \texttt{Ref l}}{H : (H_1, v) =_1 \texttt{e2} \Downarrow (H_1, \texttt{exn})}$$

$$\frac{H_1 : \texttt{e2} \Downarrow o_2 \qquad H_1 : \texttt{l} =_2 o_2 \Downarrow o}{H : (H_1, \texttt{Ref l}) =_1 \texttt{e2} \Downarrow o} \qquad \frac{\textbf{abort}\, o}{H : \texttt{l} =_2 o \Downarrow o} \qquad \frac{H_3 = H_2[\texttt{l} \leftarrow \texttt{v2}]}{H : \texttt{l} =_2 (H_2, \texttt{v2}) \Downarrow (H_3, \texttt{v3})}$$

Fig. 7: Rules for binary operators in pretty-big-step-semantics.

In this presentation, only the last rule computes, the other ones simply evaluate intermediate results and propagate errors. This avoids duplication of nearly identical reduction rules, which are frequent in the semantics. As a side effect, this increases proof automation performance as automatic tactics will not have to evaluate twice identical sub-reductions. We have found this approach to be very useful as we added constructs to the language which changed the control flow.

Figure 8 shows an example of reduction rule from the interpreter compared to the reduction rule in the semantics. The main difference between the two is that the semantics is defined as an inductive predicate, whereas the interpreter is a fixpoint. The three main cases of Figure 7 can be seen in (a) (we do not depict the abort rules). The first rule evaluates `e1` and passes the result `o1` to the second rule. The second rule extracts from its first argument the heap and result `re` of the evaluation of `e1` and uses them first to evaluate `e2`, then to pass the result `o2` to the third rule. The third rule is only defined if `re` is actually a reference. Another rule would cover the other cases, resulting in an abort. The third rule also extracts from its second argument the heap and result `r` of evaluating `e2`. It may

```
1  | red_expr_expr_assign : ∀h0 s e1 e2 o o1,
2      red_expr h0 s e1 o1 →
3      red_expr h0 s (ext_expr_assign_1 o1 e2) o →
4      red_expr h0 s (expr_assign e1 e2) o
5
6  | red_expr_ext_expr_assign_1 : ∀h0 h1 s e2 re o o2,
7      red_expr h1 s e2 o2 →
8      red_expr h1 s (ext_expr_assign_2 re o2) o →
9      red_expr h0 s (ext_expr_assign_1
10       (out_expr_ter h1 re) e2) o
11
12 | red_expr_ext_expr_assign_2 : ∀h0 h1 h2 s r l x v,
13     getvalue h1 r v →
14     h2 = update h1 l x v →
15     red_expr h0 s (ext_expr_assign_2 (Ref l x)
16       (out_expr_ter h1 r)) (out_expr_ter h2 v)
```

(a) Main rules related to assign expressed in the Coq 's pretty big step semantics.

```
1 | exp_assign e1 e2 ⇒
2   if_success (run' h0 s e1) (fun h1 r1 ⇒
3     if_is_ref h1 r1 (fun l f ⇒
4       if_success_value (run' h1 s e2) (fun h2 v ⇒
5         out_return (update h2 l f v) v)))
```

(b) The assign rule expressed in the Coq interpreter.

Fig. 8: Comparison of a rule expressed in the Coq semantics and the Coq interpreter.

then get the value `v` from `r` using `getvalue`. This function corresponds to the γ function of [GMS12]: if its argument is a reference, it returns the value stored at this reference; if its argument is a value, it returns this value. A final heap is built, updating the reference with `v`, and the returned outcome is this heap and value `v`.

The `run'` function is the interpreter's main function, set with a bound on the maximum number of reduction inferior to the current one. Each of the functions of the interpreter (see Figure 8b) `if_success_value`, `if_defined`, ..., represents one pair of reduction rule in the pretty big step presentation. Indeed in pretty-big-step, rules can often be grouped in pairs: one that deals the case where the previous execution sent an exception, and the other case that deals the normal one. For instance in Figure 7, the two first rules would be packed in one function matching the return of `e1`. Grouping the reduction rules in such functions makes the writing style of the interpreter monadic, which helps proving its correctness.

The interpreter is thus written in a continuation-style, which is relatively close to the pretty big step reduction. When the result of `run' h0 s e1` is a failure, the continuation is not evaluated and the failure is returned. Similarly, when `if_is_ref` is called at Line 3, `r1` is destructed to a reference to location `l` and field `f`, a failure being raised if it is not a reference.

Except for those minor differences, the definition of the interpreter and the reductions rules are very similar. For each predicate defined in the semantics, a function is defined in the interpreter. For instance, to `proto` (which is the predicate representation of π in the semantic file) of type `jsheap` → `field` → `loc` → `loc` → `Prop` corresponds a function `proto_comp` of type `jsheap` → `field` → `loc` → `loc`. Those duplications are needed as in proofs it is easier to manipulate inductive propositions than functions. Furthermore, as predicate representation is closer to paper definitions, it is better to have inductive predicates rather than implementations in the trusted base. This separation also allows to optimize the implementation, as long as it is proven to correspond to the predicate. For instance, Figure 9 shows that `proto_comp` is equivalent to the predicate `proto` under the assumption that their arguments are valid.

```
Lemma proto_comp_correct : ∀h f l l',
  ok_heap h →
  bound h l →
  proto_comp h f l = l' →
  proto h f l l'.
```

(a) Correctness.

```
Lemma proto_comp_complete : ∀h f l l',
  ok_heap h →
  bound h l →
  proto h f l l' →
  proto_comp h f l = l'.
```

(b) Completeness.

Fig. 9: Correctness and completeness of the function π with respect to the `proto` predicate.

2.4. Extracting the Interpreter

The extraction of the interpreter in OCaml is fairly straightforward. We extract natural numbers to `int` and Coq strings to list of characters. As the floats use in JavaScript are the same one that of OCaml, we extract floats from Flocq into OCaml floats as follows.

```
Require Import ExtrOcamlZInt.
Extract Inductive Fappli_IEEE.binary_float ⇒float [
  "(fun s →if s then (0.) else (-0.))"
  "(fun s →if s then infinity else neg_infinity)"
  "nan"
  "(fun (s, m, e) →let f = ldexp (float_of_int m) e in if s then f else -.f)"
].
Extract Constant number_comparable ⇒"(=)".
Extract Constant number_add ⇒"(+.)".
Extract Constant number_mult ⇒"( *. )".
Extract Constant number_div ⇒"(/.)".
Extract Constant number_of_int ⇒float_of_int.
```

The extracted interpreter is about 5000 loc. As efficiency is not our goal, we have not run benchmarks. We have however tested it on small programs, allowing to enforce our trust on the 600 loc semantics.

3. Proving Properties

3.1. Heap's Correctness

Let us re-consider the rules of Figure 6 for the π function, that looks for a given variable in a prototype chain. There exist some cases where this function is not defined. For instance when $l \neq$ `null`, $(l, x) \notin \mathrm{dom}(H)$, and $(l, @proto) \notin \mathrm{dom}(H)$. We have to prove that π is never called in such cases (otherwise the reduction is unsound).

To this end, we rely on a definition of heap correctness. It is defined as a record of some properties. Among those correctness condition, there are for instance `ok_heap_null` that states that the location `null` is not bound in the heap, or `ok_heap_protochain` that states that each bound location has a correct prototype chain (the field *@proto* has thus to be defined for each non-`null` element in it and no loop should appear). Similarly, a notion of scope chain correctness and of result correctness have been defined.

One important result of the JSCert project is a safety theorem, stating that this notion of correctness is conserved through the formalized JavaScript reduction rules. As we need to cover expressions, statements, and programs which all depend on each other, we write the theorem as a fixpoint, giving explicitly the decreasing argument on which to base the inductive proof.

```
Fixpoint safety_expr h s e o (R : red_expr h s e o) {struct R} :
  ok_heap h →
  ok_scope h s →
  ok_ext_expr h e →
  ok_out_expr h o
with safety_stat h s p o (R : red_stat h s p o) {struct R} :
  ok_heap h →
  ok_scope h s →
  ok_ext_stat h p →
  ok_out_prog h o
with safety_prog h s P o (R : red_prog h s P o) {struct R} :
  ok_heap h →
  ok_scope h s →
  ok_ext_prog h P →
  ok_out_prog h o.
```

This theorem is actually expressed over extended expressions, which contain some heaps or intermediary results. The `ok_ext_ext`, `ok_ext_stat`, and `ok_ext_prog` predicates state that those intermediary objects are also correct.

To better understand the statement of the theorem, here is one of the constructors of `ok_out_expr` (the other constructors deal with exceptions).

```
Inductive ok_out_expr h0 : out_expr → Prop :=
    ok_out_expr_normal : ∀h (r : ret_expr),
                         ok_heap h →
                         ok_ret_expr h r →
                         extends_proto h0 h →
                         ok_out_expr h0 (out_expr_ter h r)
```

The predicate `extends_proto` states that the implicit prototypes (the value of their fields @*proto*, not to be confused with their fields `prototype`) of objects cannot be removed through reduction: their values can change, but those fields cannot be deleted. There are indeed checks that may lead to exceptions on the deletion of some fields when calling the `delete` operator (for instance, it is not possible to delete a @*proto* field). This predicate is needed in function calls as old scope chains are restored; those old scope chains having to be still correct in the new heap, and thus have to keep their prototypes. This is ensured by lemmas such as the following.

```
Lemma ok_scope_extends : ∀h1 h2 s,
  ok_scope h1 s →
  extends_proto h1 h2 →
  ok_scope h2 s.
```

3.2. Interpreter's Correctness

The interpreter has been proven correct: each time it returns a defined result (a value or an exception), this result was correct with respect to the semantic rules. If it returns `out_bottom` because the bound it was given over the maximum number of reductions was not large enough, or any other error, then the theorem does not apply. Here is the property the theorem proves, the predicates `ret_res_expr` interfacing the data-types of the interpreter and of the semantics.

```
Definition run_expr_correct_def m := ∀h s e h' r re,
  run_expr m h s e = out_return h' r →
```

```
3   ret_res_expr r re →
4   ok_heap h →
5   ok_scope h s →
6   red_expr h s e (out_expr_ter h' re).
```

For most cases of this theorem, the proof strategy consists in splitting the goal using elimination lemmas in all the possible cases to deconstruct the equality of Line 2 until a result is returned by `run_expr`, then proving properties of intermediary results and at last folding the `red_expr` predicate. Let us consider the example of the assignment rule given in Figure 8. Three functions are nested there, thus three elimination lemmas will be used. Let us consider the one for `if_success` (see Section 2.3).

```
1  Lemma elim_if_success : ∀r0 k h r,
2    if_success r0 k = out_return h r →
3    (r0 = out_return h r ∧ ∀v, r ≠ret_res v) ∨
4      ∃r1 h0, r0 = out_return h0 (ret_res r1).
```

It takes as argument the fact the computation stops, and gives a disjunction: either `r0` was an exception or an error, or there exists a result. After applying this lemma, the goal is split in two. In one case, the result was either an error (which is not possible as Line 2 of the theorem's statement expect it to be a result), or it is an exception and we may directly apply an `abort` rule to conclude. In the other case, we get a result `r1` and a new heap, which allows us to rewrite the equality `if_success r0 (...)= out_return h' (ret_res v)` to:

```
1   R : if_is_ref h1 r1 (fun (l : loc) (f : field) ⇒
2         if_success_value (run m h1 s e2) (fun (h2 : jsheap) (v : val) ⇒
3           out_return (update h2 l f v) v))
4       = out_return h' v
```

The first step of the assignment rule's computation has now been performed and the proof continues. In this case, the fact that the new heap `h1` is correct is directly given by the safety theorem, but this is not always the case. Indeed, when some writes are performed on heaps, this last theorem does not always apply for those intermediary results. This has led to some code copied from the safety proof to the interpreter correctness proof. It may be interesting to slightly change the structure of the safety proof to factorize those steps.

3.3. Completeness

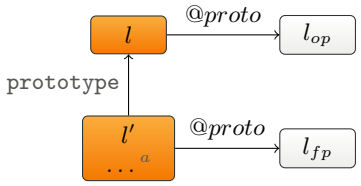
The interpreter has not yet been proven complete. The main reason is that in the inductive semantics, heap allocation is unspecified, whereas the interpreter allocates locations in a deterministic way. Let us consider the simplest reduction rule that involves some allocations in the heap: the declaration of a new unnamed function. Figure 10 shows the corresponding Coq rule and a graphical representation of its meaning. The predicate `fresh`, called twice, is defined by stating that the given location is not `null` and is not bound in the current heap, with no other restriction. In the interpreter, however, the function `fresh_for` is defined as the minimum location number not yet allocated in the current heap.

To prove completeness, one has to show that given equivalent heaps, where every location is injectively renamed, an expression reduces to the same value and to equivalent heaps. As we keep extending the semantics to take additional constructs into account, we have not yet proven this result. Going further, given a heap and a list of live locations, we could define an equivalent heap which may contain fewer locations, thus add garbage collection to the interpreter. We are currently exploring another option, which consists in making the semantics deterministic by equipping the heap with an unspecified function that deterministically returns fresh locations.

A second reason for delaying the proof of completeness concerns parsing. Indeed, as JavaScript has an `eval` operator, parsing is part of the semantics of programs and the exact parsing rules have to

```
1  | red_expr_expr_function_unnamed : ∀l l' h0
       h1 h2 s lx P,
2      fresh h0 l →
3      h1 = alloc_obj h0 l loc_obj_proto →
4      fresh h1 l' →
5      h2 = alloc_fun h1 l' s lx P l →
6      red_expr h0 s (expr_function None lx P)
           (out_expr_ter h2 l')
```

(a) Coq code to define a new function.



(b) A graphical representation of it.

[a] This location now contains the current scope chain and the function body P.

Fig. 10: Allocating new unnamed function.

be formalized. As first approximation, we could define a partial parsing algorithm which may wrongly reject some programs. This approach would obviously not be complete, but it would also not be correct, as a parse error in the context of the execution of an `eval` operator is a runtime exception, which we need to model. Thus the property of a string not being parseable must be formally defined and we have to aim for completeness to correctly support the `eval` operator. The problem with JavaScript parsing is that it is quite complex. For instance, semicolons are not always mandatory, and the rules expressing *automatic semicolon insertion* use backtracking, exceptions, and further tests. This precludes the use of classical parser technology. We plan on working on this issue when most of the language is formalized, relying on and extending existing work on parser validation [JPL12].

Finally, a practical way to test for completeness would be to confront our interpreter to existing test suites. We are also planning on doing so as soon as we have covered the language sufficiently.

4. Related and Future Work

4.1. Related Work

The work on JSCert and on the interpreter could not have been done without relying on the formalization by Maffeis et al. [GMS12,MMT11,MMT08]. There have been other attempts to give formal accounts of JavaScript's semantics, which we now review.

A common approach is to define a simple formal language or calculus on which properties are more easily proved, then write a desugaring function from JavaScript to the simpler language. The λ_{JS} project [GSK10] follows such an approach in a small-step setting. They have proven, in Coq, that the produced terms never get stuck (they either can reduce, are a value, or are an error). However, they only relate the JavaScript terms and the desugared version through testing. To this end, they implemented an interpreter for λ_{JS}. In [CHJ12], Chugh et al. present DJS, an extension of their calculus for dynamic languages [CRJ12], with features to mimic JavaScript constructions such as imperative updates, prototype inheritance, and arrays. As in λ_{JS}, they use desugaring to go from JavaScript to DJS, with no formal claim of correctness.

Other approaches focus more on the formalization of how JavaScript interacts with the browser, from network communication to the DOM representation [Boh12,YCIS07]. These works do not aim at covering the whole language and present very promising extensions to our semantics and interpreter, once we have finished formalizing the core language.

Finally, many other works have formalized part of JavaScript in order to develop static or dynamic analyses, focusing on particular aspects of the language and its runtime. In particular, API access has been studied based on a Datalog model of JavaScript [TEM+11]; Hedin and Sabelfeld

have written a big-step paper semantics of JavaScript to dynamically track information flow [HS12]; Luo and Rezk have proposed a decorated semantics to prove correctness and security properties of a JavaScript to JavaScript compiler for mashups [LR12]. It is our hope that a complete and precise formalization will help avoid further duplication of effort.

4.2. Extensions of the Interpreter

Our goal is to include all of the core specification from ES3. We are currently working on adding `eval` and type conversion to the semantics and the interpreter. We will then turn to some primitive objects and features, such as arrays, to test the interpreter against more realistic programs. We will then consider the additions of ES5, in particular *strict mode* and accessors.

Further, we want to go beyond the specification. In practice, most real world interpreters do not strictly follow it, typically by accepting reads or even writes on some (theoretically) unaccessible fields (such as the implicit prototype @*proto*, often called `__proto__` by those non-standard interpreters). We plan to add such features in the formalization (and thus in the interpreter) in a modular way. To this end, we will parameterize the interpreter by some flags describing which standards it should follow (ES3, ES5, Firefox's one, etc.). It will then be possible to prove security of a given program for a given non-standard browser.

Finally, we plan to use the interpreter to test the semantics on real JavaScript test suites. This will provide additional trust on the semantics and thus everything that depends on it (such as certified code analysis). Interestingly enough, writing and proving the interpreter has actually helped uncovering some missing cases and misconceptions in the Coq semantics. These bugs were fairly subtle, most of them being a confusion between extended expressions `ext_expr` (introduced by the pretty big step presentation) and expressions `expr`, which is difficult to verify by hand.

Conclusion

We have presented the design and implementation of a JavaScript interpreter proven correct in relation with the semantics developed in the JSCert project. The main motivation for developing a formal semantics for JavaScript is twofold: JavaScript has become pervasive in web development, and its semantics is fairly complex. As a result, providing strong guarantees for JavaScript programs is difficult yet would have a significant impact.

The proof of correctness of the interpreter has been done in the Coq proof assistant, and the implementation is extracted in OCaml from the development. Many features of JavaScript are supported, including prototype-based inheritance, explicit scope manipulation, and exceptions. The development is available on the JSCert web site [BCF+12].

The completeness of the interpreter has not yet been proven, for two main reasons. First, it requires relating the undeterministic heap allocation of the semantics to the deterministic one of the interpreter. Second, it needs to formally specify the parsing of strings to correctly and completely model the `eval` operator.

We believe that our three-tiered approach—write a semantics following closely the specification, independently write and prove correct an interpreter, and test the interpreter—yields a good level of trust in the formal semantics to base further works on it. We have started to investigate the formal development of static analyses of JavaScript programs as a continuation of this work.

Bibliographie

A+99. European Computer Manufacturers Association et al. Ecmascript language specification, 1999.

BCF+12. M. Bodin, A. Charguéraud, D. Filaretti, P. Gardner, S. Maffeis, D. Naudziuniene, A. Schmitt, and G. Smith. Jscert: Certified javascript. http://jscert.org/, 2012.

BM10. S. Boldo and G. Melquiond. Flocq: A Unified Library for Proving Floating-point Algorithms in Coq. Soumis à ARITH-20 (2011), October 2010.

Boh12. A. Bohannon. *Foundations of Web Script Security*. PhD thesis, University of Pennsylvania, 2012.

Cha10a. A. Charguéraud. The optimal fixed point combinator. In Matt Kaufmann and Lawrence C. Paulson, editors, *Proceeding of the first international conference on Interactive Theorem Proving (ITP)*, volume 6172 of *LNCS*, pages 195–210. Springer, 2010.

Cha10b. A. Charguéraud. TLC: a non-constructive library for coq based on typeclasses. http://www.chargueraud.org/softs/tlc/, 2010.

Cha12. A. Charguéraud. Pretty-big-step semantics. In *Proceedings of the 22nd European Symposium on Programming (ESOP)*, 2013. To appear.

CHJ12. R. Chugh, D. Herman, and R. Jhala. Dependent types for javascript. In *Proceedings of OOPSLA 2012*, 2012.

CRJ12. R. Chugh, P. M. Rondon, and R. Jhala. Nested refinements: a logic for duck typing. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '12, pages 231–244, New York, NY, USA, 2012. ACM.

Eny12. Enyo. Enyo web site. http://enyojs.com/, 2012.

GMS12. P.A. Gardner, S. Maffeis, and G.D. Smith. Towards a program logic for javascript. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 31–44. ACM, 2012.

GSK10. A. Guha, C. Saftoiu, and S. Krishnamurthi. The essence of javascript. *ECOOP 2010–Object-Oriented Programming*, pages 126–150, 2010.

HS12. D. Hedin and A. Sabelfeld. Information-flow security for a core of javascript. In *Proceedings of the 25th IEEE Computer Security Foundations Symposium, CSF 2012*, pages 3–18, Cambridge, MA, USA, June 2012.

JPL12. J.-H. Jourdan, F. Pottier, and X. Leroy. Validating lr(1) parsers. In *Proceedings of the 21st European conference on Programming Languages and Systems*, ESOP'12, pages 397–416, Berlin, Heidelberg, 2012. Springer-Verlag.

LR12. Z. Luo and T. Rezk. Mashic compiler: Mashup sandboxing based on inter-frame communication. In IEEE, editor, *Proceedings of 25th IEEE Computer Security Foundations Symposium*, pages 157–170, Cambridge, MA, USA, June 2012.

MMT08. S. Maffeis, J. Mitchell, and A. Taly. An operational semantics for javascript. *Programming Languages and Systems*, pages 307–325, 2008.

MMT11. S. Maffeis, J.C. Mitchell, and A. Taly. An operational semantics for javascript. http://jssec.net/semantics/, 2011.

Moz12. Mozilla. B2g wiki. https://wiki.mozilla.org/B2G, 2012.

SGL06. M. Serrano, E. Gallesio, and F. Loitsch. Hop: a language for programming the web 2.0. In *Proceedings of the First Dynamic Languages Symposium*, pages 975–985, Portland, OR, USA, October 2006. ACM.

SO08. M. Sozeau and N. Oury. First-class type classes. *Theorem Proving in Higher Order Logics*, pages 278–293, 2008.

TEM+11. A. Taly, Ú. Erlingsson, J.C. Mitchell, M.S. Miller, and J. Nagra. Automated analysis of security-critical javascript apis. In *Security and Privacy (SP), 2011 IEEE Symposium on*, pages 363–378. IEEE, 2011.

VB11. J. Vouillon and V. Balat. From bytecode to javascript: the js_of_ocaml compiler. unpublished, 2011.

YCIS07. D. Yu, A. Chander, N. Islam, and I. Serikov. Javascript instrumentation for browser security. In *Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '07, pages 237–249, New York, NY, USA, 2007. ACM.

Zak11. A. Zakai. Emscripten: an llvm-to-javascript compiler. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, SPLASH '11, pages 301–312, New York, NY, USA, 2011. ACM.

Session du groupe de travail MTV^2

Méthodes de test pour la validation et la vérification

Towards a Common Specification Language for Static and Dynamic Analysis of C Programs*

Extended Astract

Mickaël Delahaye[1], Nikolai Kosmatov[2] and Julien Signoles[2]

[1] UJF-Grenoble 1, LIG, UMR 5217, 38041 Grenoble, France
firstname.lastname@imag.fr
[2] CEA, LIST, Software Safety Laboratory, PC 174, 91191 Gif-sur-Yvette France
firstname.lastname@cea.fr

Introduction

A usual input of software verification tools includes a program and its (partial) specification. Testing tools need at least a *precondition* (or *test context*) specifying admissible input data on which the program should be tested, and may require an *oracle*, deciding if the results of the execution on a given test are correct. Detecting potential runtime errors by abstract interpretation also needs a precondition to improve its precision. Tools for proof of programs require a formal specification (or *contract*) with pre/postconditions, loop invariants, *etc*. Although the specification is extremely important for the verification process, its format varies from one tool to another, especially between static and dynamic analysis tools. That makes it difficult to combine them in a completely automatic way.

Recent research showed that combinations of static and dynamic analysis can be beneficial for software verification. One concrete example is SANTE [1] which efficiently combines the value analysis plug-in of FRAMA-C[3] [2], a platform dedicated to analysis of C programs, and the structural test generation tool PATHCRAWLER [3] for detection of runtime errors in C programs. While all static analyzers of FRAMA-C share a common specification language, called ACSL [4], PATHCRAWLER requires a precondition specified in another format and an oracle defined by a C function. Rewriting the precondition of the target C function in the PATHCRAWLER format remains the only manual step of the SANTE method.

Executable ANSI/ISO C Specification Language: E-ACSL

This talk presents an overview of E-ACSL [5,6], an expressive sub-language of ACSL that can be translated into C, compiled and used as executable specification. We also describe its automatic translator E-ACSL2C into C [7].

The E-ACSL design brings several benefits. To the best of our knowledge, E-ACSL is the first formal behavioral specification language for C that builds a bridge between static and dynamic analysis tools and avoids manual rewriting of a formal program

* This work has been partially funded by the FUI9 'Hi-Lite' project.
[3] http://frama-c.com

specification for testing. Second, choosing a sub-language of ACSL has the advantage of being supported by existing FRAMA-C analyzers. Third, translating into C rather than into a specific format of a particular tool allows the usage by other analysis tools for C. Fourth, the possibility to observe the status of an annotation during a concrete execution may be very helpful for writing a correct specification of a given program, *e.g.* for later program proving. Finally, an executable specification makes it possible to check at runtime assertions that cannot be verified statically, so linking monitoring and static analysis tools.

We emphasize particular issues related to specific keywords, quantifications, mathematical integers, memory-related annotations and undefined terms in E-ACSL. We present our solutions for these issues. Most of them are already implemented and available in the current version of the E-ACSL2C translator. Moreover, we identifiy potential drawbacks in the current translation and propose several improvements to make it usable in practice.

Specifications translated by E-ACSL2C are also usable by testing tools for C programs. That avoids to manually rewrite the specification in another specific format for such tools. We illustrate this approach with PATHCRAWLER which automatically handles a translated specification. The experiments done with the combined method SANTE [1] showed that SANTE is more precise than a static analyzer and more efficient in terms of time and number of detected bugs than test generation alone. The present work offers a common specification language for static and dynamic analysis tools and help to develop and better automatize their combinations. Future work includes finalizing the development of E-ACSL2C, its integration into the SANTE tool and further exploration of combined techniques for software verification.

References

1. Chebaro, O., Kosmatov, N., Giorgetti, A., Julliand, J.: Program slicing enhances a verification technique combining static and dynamic analysis. In: the ACM Symposium on Applied Computing (SAC 2012), ACM (2012) 1284–1291
2. Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C, a program analysis perspective. In: the 10th Int. Conference on Software Engineering and Formal Methods (SEFM 2012). Volume 7504 of LNCS., Springer (2012) 233–247
3. Botella, B., Delahaye, M., Hong-Tuan-Ha, S., Kosmatov, N., Mouy, P., Roger, M., Williams, N.: Automating structural testing of C programs: Experience with PathCrawler. In: the 4th Int. Workshop on Automation of Software Test (AST 2009), IEEE Computer Society (2009) 70–78
4. Baudin, P., Filliâtre, J.C., Hubert, T., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C Specification Language, v1.6. (September 2012) URL: http://frama-c.com/acsl.html.
5. Signoles, J.: E-ACSL: Executable ANSI/ISO C Specification Language. (January 2012) URL: http://frama-c.com/download/ e-acsl/e-acsl.pdf.
6. Delahaye, M., Kosmatov, N., Signoles, J.: Common specification language for static and dynamic analysis of c programs. In: the ACM Symposium on Applied Computing (SAC 2013), ACM (2013) To appear.
7. Signoles, J.: E-ACSL Version 1.5-4. Implementation in Frama-C Plug-in E-ACSL version 0.1. (January 2012) URL: http://frama-c.com/download/e-acsl/ e-acsl-implementation.pdf.

Vers de outils de test formellement vérifiés : de la génération de tests à la résolution de contraintes

Matthieu Carlier[1], Catherine Dubois[1,2], and Arnaud Gotlieb[2,3]

[1] CEDRIC-ENSIIE, Évry, France
[2] Inria, France
[3] Certus V&V Center, SIMULA RESEARCH LAB., Lysaker, Norway

Introduction

De nombreux outils de génération de tests (tels que Pex, Sage, Gatel, Path-Crawler, Euclide) utilisent les contraintes et reposent donc sur des solveurs de contraintes SAT, SMT ou solveurs à domaines finis. Cependant ces outils de test mettent souvent en œuvre des techniques ou heuristiques complexes pour générer les tests, et produisent malheureusement des résultats qui peuvent être incorrects. De manière concrète, ces outils peuvent parfois annoncer avoir généré un jeu de test qui couvre complètement un critère de test, alors qu'il n'en est rien. La confiance dans ces outils de test devient cruciale dans le cadre du développement des programmes critiques.

Nous adressons le problème de la vérification des outils de génération de tests à base de contraintes au travers de deux questions. La première concerne la capture du comportement du programme - ou des propriétés à vérifier ou encore des objectifs de couverture - par un système de contraintes. Comment assurer que le comportement a bien été traduit ? La deuxième question concerne la correction du solveur de contraintes utilisé. Si ce dernier répond UNSAT, comment être sûr que le système n'a effectivement pas de solutions ? Nous proposons de répondre aux deux questions en utilisant l'assistant à la preuve Coq, donc en développant ces outils dans Coq et en démontrant qu'ils sont corrects vis-à-vis des propriétés recherchées. Plus précisément, nous présentons une approche où la sémantique d'un programme Focalize est préservée lors de sa traduction en un ensemble de contraintes (voir pour plus de détails [3]). Nous présentons ensuite brièvement le solveur à domaines finis formellement vérifié, développé dans cette optique [2].

Préservation de la sémantique dans FocalTest

L'outil FocalTest [1] intégré à l'environnement Focalize (`http://focalize.inria.fr`) permet de générer et exécuter automatiquement des jeux de test pour des programmes fonctionnels (écrits dans la langage Focalize) en mettant l'accent sur une ou plusieurs propriétés à satisfaire de la forme *précondition* ⇒ *conclusion*. Nous recherchons alors des données de test qui satisfont la précondition, invalident la précondition et aussi qui couvrent le critère de test MC-DC. Ces données sont trouvées en résolvant le système de contraintes qui traduit le comportement de la précondition et des fonctions utilisées dans celle-ci. Quant au verdict de test il est donné par l'évaluation de la conclusion.

Le langage de contraintes utilisé ici est un langage qui incorpore la notion de prédicat à la prolog et qui peut manipuler des variables à domaines finis mais aussi des variables ayant des types algébriques (définis par des constructeurs).

Nous avons prouvé la préservation sémantique de la traduction des programmes Focalize (et de la précondition de la propriété sous test) en systèmes de contraintes. La traduction s'effectue en deux temps : les programmes sont tout d'abord mis sous une forme monadique qui facilite la deuxième étape, à savoir la traduction en contraintes. Nous avons dons défini en Coq la sémantique opérationnelle d'un sous-ensemble du langage d'entrée, la sémantique opérationnelle du langage de contraintes (sous forme d'un prédicat qui décrit la notion de solution d'un système de contraintes), puis nous avons défini la fonction de traduction et prouvé sa correction et sa complétude. La fonction de traduction a pu être extraite du développement Coq, fournissant ainsi un outil de traduction écrit en OCaml, formellement vérifié.

Solveur de contraintes formellement vérifié

Nous avons développé un solveur de contraintes binaires à domaines finis prouvé correct et complet à l'aide du système Coq. Correct signifie ici que toute solution retournée satisfait bien les contraintes. De plus si le solveur répond UNSAT, le système de contraintes n'a en effet pas de solution. Le solveur obtenu est purement fonctionnel, écrit en OCaml, extrait du développement en Coq. Un point clé de ce solveur est sa généricité : il est paramétré par le langage de contraintes défini via une fonction d'interprétation des contraintes. Il met en œuvre l'algorithme classique AC3 [4], au cœur de nombreux solveurs existants, reposant sur la notion de consistance locale dite consistance d'arc.

Plus exactement le solveur est décrit à l'aide de trois processus entrelacés : filtrage, propagation, énumération. Le filtrage élimine, pour une contrainte, les valeurs incohérentes des variables de la contrainte (i.e. celles qui ne pourront faire partie d'aucune solution), la propagation détermine les contraintes qu'il faut filtrer à nouveau, impactées par l'élagage précédent. Ces deux étapes sont complétés d'une énumération des valeurs d'un domaine si besoin, ce qui amène de nouvelles incohérences à filtrer et propager, etc. jusqu'à l'obtention d'une solution ou de la découverte qu'aucune solution n'existe. Nous avons implanté ces trois processus sous la forme de fonctions Coq puis nous avons démontré leur correction et complétude. La propriété de correction du filtrage et de la propagation consiste à montrer que la consistance locale est assurée. La propriété de complétude consiste à montrer que l'application de ces étapes (qui réduisent les domaines des variables) ne perdent pas de solution.

Références

1. M. Carlier, C. Dubois, and A. Gotlieb. Constraint reasoning in focaltest. In *Int. Conf. on Soft. and Data Tech. (ICSOFT'10)*, Athens, Jul. 2010.
2. M. Carlier, C. Dubois, and A. Gotlieb. A certified constraint solver over finite domains. In *Formal Methods (FM 2012)*, volume 7436 of *LNCS*, pages 116–131, Paris, 2012.

3. M. Carlier, C. Dubois, and A. Gotlieb. A first step in the design of a formally verified constraint-based testing tool : Focaltest. In *Proc. of 6th Int. Conf. on Test&Proofs (TAP'12)*, May. 2012.
4. A. Mackworth. Consistency in networks of relations. *Art. Intel.*, 8(1) :99–118, 1977.

A Symbolic Framework for the Conformance Checking of Value-Passing Choreographies*

Huu Nghia Nguyen[1], Pascal Poizat[1,2], Fatiha Zaïdi[1]

[1] LRI; Univ. Paris-Sud, CNRS, Orsay, France
[2] Univ. Évry Val d'Essonne, Evry, France
{huu-nghia.nguyen, pascal.poizat, fatiha.zaidi}@lri.fr

Abstract. Choreography conformance checking aims at verifying whether a set of local specifications match a global one. This activity is central in both top-down and bottom-up development processes for distributed systems. Such systems usually collaborate through information exchange, thus requiring value-passing choreography languages and models. As an alternative, we propose a conformance checking framework based on symbolic models and an extension of the symbolic bisimulation equivalence. This enables one to take into account value passing while avoiding state space explosion issues. Our framework is fully tool supported[3].

Context and Issues. *Choreography* is the description with a global perspective of interactions between *roles* played by *peers* (services, organizations, humans) in some collaboration. One key issue in choreography-based development is checking the *conformance* of a set of local specifications (role requirement, peers description) wrt. the global one (choreography). This issue naturally arises both in bottom-up and in top-down development processes, and is also a cornerstone for realizability checking. The conformance relation should not be too strict. It should support choreography refinement, *e.g.*, with peers and interactions being added in the implementation by the service architect in order to enforce the specification. Furthermore, entities in a distributed system usually exchange information, *i.e.*, data, while interacting. However, most of the conformance checking techniques abstract value-passing or bound the domains for the exchanged data. This is known to yield over-approximation issues, *e.g.*, false negatives in the verification process. Consequently, data should be supported in choreography specifications, in the local specifications, and in the conformance relation.

The Framework. We have proposed a formal framework[4], as depicted in Figure 1, for checking the conformance of a set of local specifications with reference to a choreography one. It takes as input a global specification C, with m roles, and also an implementation specification I, given as $n \geq m$ local specifications. The case when $n > m$ denotes, *e.g.*, an implementation where some peers have

* This work is supported by the ANR PIMI project (ANR-2010-VERS-0014-03).
[3] Our tool is freely available at http://www.lri.fr/~nhnghia/sbbc/
[4] An extension of this paper appears in [1]

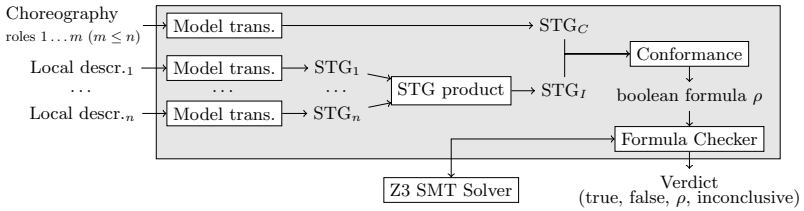


Fig. 1. Architecture of our framework

been added to make a choreography realizable. We propose a specification and description language for describing these inputs. The language is based on process algebras for choreography [2] for addressing both the global and the local perspective over distributed systems, and for supporting information exchange and data-related constructs (conditional and loop constructs).

The inputs are transformed into Symbolic Transition Graphs (STGs). An STG is a transition system where each transition has the form $s \xrightarrow{[\phi]\ \alpha} s'$ in which ϕ is a boolean expression that has to hold for the transition to take place, and α is an event. The product of STGs and the restriction to actions in C are used to retrieve a unique STG for I, thus yielding two STGs to compare: one for C ($\mathcal{C}$) and one for I ($\mathcal{I}$). We check if $\mathcal{I}$ conforms to $\mathcal{C}$. Our conformance relation is a symbolic extension of branching bisimulation [3]. This generates the largest boolean formula ρ such that the initial states of $\mathcal{I}$ and $\mathcal{C}$ are conformance related. The formula ρ is analysed by the Z3 SMT solver to reach a conformance verdict. This can be "always true" or "always false", "always" meaning whatever the data values exchanged between peers are. Sometimes we can have conformance only for a subset of these values. Going further than pure true/false conformance, our framework thus allows to compute the *largest constraint on data values*, ρ, that would yield conformance. The inconclusiveness verdict is emitted when complex constraints cause the solver to return a timeout.

Future Works. We advocate that once a choreography projection function supporting data is defined, then our framework could be used not only for conformance checking but also for realizability checking. This is our first perspective. A second perspective is to extend our framework with non-limited assignment and asynchronous communication.

References

1. Nguyen, H.N., Poizat, P., Zaïdi, F.: A Symbolic Framework for the Conformance Checking of Value-Passing Choreographies. In: ICSOC'2012. (2012) 525–532
2. Qiu, Z., Zhao, X., Cai, C., Yang, H.: Towards The Theoretical Foundation of Choreography. In: Proc. of WWW '07. (2007)
3. Van Glabbeek, R., Weijland, W.: Branching Time and Abstraction in Bisimulation Semantics. Journal of the ACM **43**(3) (1996) 555–600

Session du groupe de travail RIMEL

Rétro-Ingénierie, Maintenance et Evolution des Logiciels

Feature Identification From the Source Code of Product Variants

Tewfik Ziadi[1], Luz Frias[2], Marcos Aurélio Almeida da Silva[1] and Mikal Ziane[1]

[1]UMR CNRS 7606, LIP6
Université Pierre et Marie Curie
4 Place Jussieu, Paris, France
{Surname.Name}@lip6.fr
[2] INDRA,
Madrid Spain
luzfrias@gmail.com@lip6.fr

Software Product Lines aim at decreasing development cost and time by developing a family of systems rather than one system at a time. A *Software Product Line* (SPL) is a set of software-intensive systems sharing a common, managed set of features that satisfies the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way [3].

Software Product Lines focus on capturing commonality and variability between several software products belonging to the same domain [3]. Commonality gathers assumptions that are true for all product members while variability concerns assumptions about how individual product members differ.

Software Product Line Engineering (SPLE) [3] can be implemented top-down: variability is first specified in what is called the *feature model* (FM) and then products are derived. This top-down process is especially interesting to create new product lines. In practice however, software product lines(SPL) are often set up after the implementation of several similar product variants using ad hoc reuse techniques such as copy-paste-modify [4].

In recent years, a lot of work [1,5,2] has addressed the identification of features. However, most of that work takes as input textual requirements [2] or architectural artefacts [1,5]. The reverse engineering of features from the source code is seldom considered.

In this work, we advocate that if the source code of the product variants is maintained and available, it is possible to investigate a bottom-up process to identify the features and at least a starting point for the feature model from product variants. This allows capitalizing the common features between the existing products and it thus facilitates their maintenance. In addition, combining the identified features may lead to the production of new products reusing the top-down process of SPLE. We propose a three-step approach to feature identification from the source code of product variants [6].

– In the first step, a model is reverse engineered from the source code of each product. The idea is to reduce the noise induced by spurious differences in the various implementations of the same feature. Each product model is then decomposed into a set of atomic pieces.

– The second step relies on an algorithm that produces feature candidates. The algorithm identifies pieces of software that appear identical in the available products from the high-level viewpoint induced by the first step.
– The third step manually prunes the non relevant candidates that may appear for several reasons including when the level of abstraction of the model does not hide all the spurious differences in the implementations of features.

This approach has been implemented and several experiments have been conducted as a preliminary evaluation. The main advantage of our approach is that it provides a quick automatic front-end to avoid most of the tedious tasks of feature identification from source code.

References

1. Acher, M., Cleve, A., Collet, P., Merle, P., Duchien, L., Lahire, P.: Reverse Engineering Architectural Feature Models. `http://hal.inria.fr/inria-00614984/en/`
2. Alves, V., Schwanninger, C., Barbosa, L., Rashid, A., Sawyer, P., Rayson, P., Pohl, C., Rummler, A.: An exploratory study of information retrieval techniques in domain analysis. In: SPLC. pp. 67–76. IEEE Computer Society (2008)
3. Clements, P., Northrop, L.: Software product lines: practices and patterns. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2001)
4. Northrop, L.: A framework for software product line practice -version 5.0. Web http://www.sei.cmu.edu/productlines/tools/framework/, Software Engineering Institute (2011)
5. Ryssel, U., Ploennigs, J., Kabitzsch, K.: Automatic variation-point identification in function-block-based models. In: Visser, E., Järvi, J. (eds.) GPCE. pp. 23–32. ACM (2010)
6. Ziadi, T., Frias, L., da Silva, M.A.A., Ziane, M.: Feature identification from the source code of product variants. In: Mens, T., Cleve, A., Ferenc, R. (eds.) CSMR. pp. 417–422. IEEE (2012)

Using Architectural Patterns to Dene Architectural Decisions

Minh Tu Ton That, Salah Sadou, and Flavio Oquendo

IRISA
Université de Bretagne Sud
Vannes, France
{minh-tu.ton-that,Salah.Sadou,Flavio.Oquendo}@univ-ubs.fr

1 Introduction

One of the major problems of software development lies in the "maintenance and evolution" stage. Indeed, given the high costs associated with this stage (about 80% of the total cost), it becomes important to find a solution to reduce them. The main factors of this problem are the non-compliance with established practices and the lack of explicitness of the choices made throughout the development process.

If we are in the first case and we need to apply an evolution, we must first rebuild what has already been improperly built. Applying an evolution on a poorly constructed system can only make it more complex and ultimately not able to evolve further. In the second case, the system is well developed, except that the intentions behind each choice are not explicit. There are always different solutions to achieve a change, but some of them may be in contradiction with certain implicit intentions. Moreover, it can take several steps between the creation of the contradiction and its detection. This requires undoing what has already been built, resulting in an important additional costs.

The first factor of the problem cited above may be avoided by human or automatic controls to check compliance with good practices. To avoid the second factor, we need to make explicit and exploitable intentions that lie behind each choice. The explicitness must begin at the software architecture definition stage. In this paper we will focus on this last point.

The intention associated with an architectural choice is designated in the literature by the term "Architectural Decision" (AD). Thus, the objective is to define the links that bind the components of AD: the property identification, the involved architectural elements and the rules defining the property. A formalization of this link serves to automatically check that an evolution does not conflict with the choices already made. Several studies have already been made to define such links. Proposed solutions usually consist of elements added to the architecture (constraints, specification, etc.) to establish the link. Although these elements indicate the presence of ADs, they do not encourage the architect to use the best solutions and/or good practices. Moreover, the added elements are often described using a language that is different from the architecture description language (ADL) used for the architecture description. So, understanding

the architecture requires a review of different elements from different languages, which complicates the task. As understanding the architecture is a step prior to its evolution, its complication undoubtedly induces a significant cost.

2 General Approach

The main idea behind our work is the leverage of architectural patterns as forms of AD representation. AD documentation in our approach falls into three steps: AD creation, AD integration and AD verification. *Decision creation* consists in the specification of an AD made to an architectural model. A decision could be specific to a project or reusable within different projects. Architectural decisions could be well-known architectural patterns which are lessons learned from many previous works or decisions that have high potential to be reused in an enterprise. *AD integration* is a step in which architects link ADs with affected elements in the architectural model. During the *AD verification* step, the architectural model is checked whether it complies with the integrated ADs.

On the purpose of automating the process of AD documentation, we use the Model Driven Architecture (MDA) approach. Each artifact is considered as a model conforming to its meta-model in order to create a systematic process thanks to model transformations and leverage existing MDA techniques (e.g. conformity verification).

3 Conclusion

Keeping track of ADs made on a system is very important to avoid degrading it during its evolution. Although there existed a lot of works focusing on documenting ADs, the automation of AD checking during the development of the architecture is still an open issue.

In this paper, we propose to document ADs in the form of formalized patterns. This approach helps guarantee the existence of ADs not only in syntactic aspect but also in some semantic aspects. With the presented approach, the purpose of AD checking, which is an error-prone task, was automated. Besides, the reusability characteristic of AD is also taken into consideration since we leverage a language-independent AD creation mechanism. We also implemented a tool to realize our approach. We utilize the case of SOA to validate our approach but it is thoroughly relevant to other types of ADLs.

Feature Mining From a Collection of Software Product Variants

Rafat AL-msie'deen[1], Abdelhak D. Seriai[1], Marianne Huchard[1], Christelle Urtado[2], Sylvain Vauttier[2] and Hamzeh Eyal Salman[1]

[1] LIRMM / CNRS & Montpellier 2 University, Montpellier, France
{Al-msiedee, Abdelhak.Seriai, huchard, eyalsalman}@lirmm.fr
[2] LGI2P / Ecole des Mines d'Alès, Nîmes, France
{Christelle.Urtado, Sylvain.Vauttier}@mines-ales.fr

1 Reverse Engineering Software Product Lines

Similarly to car manufacturers who propose a full range of cars with common characteristics and numerous variants and options, software development might entail to segment users' needs and propose to them a software family to choose from. Such software family is called a *software product line* (SPL) [1]. A SPL is usually characterized by two sets of features: the features that are shared by all products in the family, called the *SPL's commonalities*, and, the features that are shared by some, but not all, products in the family, called the *SPLs variability*. These two sets define the mandatory and optional parts of the SPL. *Software product line engineering* (SPLE) focuses on capturing the commonalities and variabilities between several software products that belong to the same family.

In order to provide a more subtle description of the possible combinations of optional features (*e.g.*, some optional feature might exclude another and require a third one), SPLs are usually described with a *de-facto* standard formalism called a *feature model*. A feature model characterizes the whole software family. It defines all valid feature sets, also called *configurations*. Each valid configuration represents a specific product, either it be an existing product or a valid product-to-be.

Software product variants are seldom developed in a disciplined way from scratch. Alternatively, *ad hoc* reuse techniques such as copy-paste-modify are used on the software's code until some point where the need to discipline the development by adopting a SPLE approach raises. Expected benefits are to improve product maintenance, ease system migration, and the extracted features may lead to the production of new products. In order to capitalize from the existing code, reverse engineering is needed but manual analysis of the existing software product variants to discover their features is time-consuming, error-prone, and requires substantial efforts. Automating feature mining from source code would be of great help.

In literature, surprisingly, the reverse engineering of features (or feature model) from source code is seldom considered [2]. Existing approaches mine features from a single software product variant, while we think it is necessary to consider all available variants at a time [3].

2 A Three Step Process to Mine Features from Code

Feature location in OO source code consists in identifying the object-oriented building elements (OBEs) that implement a particular feature across software product variants. The OBE we consider are packages, classes, attributes, methods and their body. We assume that a feature can be mapped to one and only one set of OBEs: each feature has a unique implementation for the whole product family.

In order to mine features from the OO source code of software variants, we propose a three step process and rely on both Formal Concept Analysis (FCA) [4] and Latent Semantic Indexing (LSI) [5] techniques. Our approach:

1. extracts OBEs from each software product variant by parsing its code.
2. uses FCA to build a lattice from OBEs and software product variants that hierarchically groups OBEs from the software product variants into disjoint, minimal partitions. This classification provides us with two OBE sets: Common OBEs (that are shared by all variants and can be found on the top node of the lattice) and variable OBEs (that are shared by several but not all variants and appear at the bottom of the lattice).
3. clusters OBEs into features. Each OBE set is analyzed using LSI and FCA techniques to mine the optional and mandatory features based on the lexical similarity between OBEs.

We have implemented this three step approach and evaluated its produced results on a collection of ten ArgoUML products. The results showed that most of the features were identified [6]. In our future work, we plan to combine both textual and semantic similarity measures to be more precise in determining feature implementation. We also plan to use the mined common and variable features to automate the building of the studied software family's feature model.

References

1. AL-Msie'deen, R., Seriai, A.D., Huchard, M., Urtado, C., Vauttier, S., Salman, H.E.: An approach to recover feature models from object-oriented source code. In: Actes de la Journée Lignes de Produits 2012, Lille, France (Novembre 2012) 15–26
2. AL-Msie'deen, R., Seriai, A.D., Huchard, M., Urtado, C., Vauttier, S., Salman, H.E.: Survey: reverse engineering feature model/features from different artefacts. http://www.lirmm.fr/Survey (2013) [Online; accessed 24-January-2013].
3. Dit, B., Revelle, M., Gethers, M., Poshyvanyk, D.: Feature location in source code: a taxonomy and survey. Journal of Software: Evolution and Process (2012) 5395
4. Ganter, B., Wille, R.: Formal Concept Analysis, Mathematical Foundations. Springer-Verlag (1999)
5. Marcus, A., Maletic, J.I.: Recovering documentation-to-source-code traceability links using latent semantic indexing. In: Proceedings of the 25th International Conference on Software Engineering. ICSE '03, Washington, DC, USA, IEEE Computer Society (2003) 125–135
6. AL-Msie'deen, R., Seriai, A.D., Huchard, M., Urtado, C., Vauttier, S., Salman, H.E.: ArgoUML case study. http://www.lirmm.fr/CaseStudy (2013) [Online; accessed 23-January-2013].

Table ronde : Green Computing

Table ronde : Green Computing

Animateurs

Jean-Marc Menaut (LINA, Inria, Ecole des Mines de Nantes)
Romain Rouvoy (LIFL, Inria, U. Lille)

Participants

Alain Anglade (ADEME)
Laurent Lefevre (Inria)
Frédéric Bordage (Alliance Green IT)
Olivier Philippot (Kaliterre)

Résumé :
Cette table ronde aborde les problématiques liées à l'empreinte énergétique des logiciels. En invitant des acteurs majeurs du domaine, issus de différentes communautés qu'elles soient académiques ou industrielles, pour aborder différents sujets autour du Green computing, nous souhaitons mettre en évidence la complexité et l'étendue de ce défi pour notre société. En particulier, nous nous discuterons de la place que peut avoir le génie logiciel pour contribuer efficacement à la réduction de l'empreinte énergétique des technologies de l'information et de la communication.

Prix de thèse du GDR Génie de la Programmation et du Logiciel

Continuation-Passing C : Transformations de programmes pour compiler la concurrence dans un langage impératif

Auteur : Gabriel Keirneis (University of Cambridge)

Résumé :

La plupart des programmes informatiques sont concurrents : ils doivent effectuer plusieurs tâches en même temps. Les threads et les événements sont deux techniques usuelles d'implémentation de la concurrence. Les événements sont généralement plus légers et efficaces que les threads, mais aussi plus difficiles à utiliser. De plus, ils sont souvent trop limités ; il est alors nécessaire d'écrire du code hybride, encore plus complexe, utilisant à la fois des threads ordonnancés préemptivement et des événements ordonnancés coopérativement.

Nous montrons dans cette thèse que des programmes concurrents écrits dans un style à threads sont traduisibles automatiquement en programmes à événements équivalents et efficaces par une suite de transformations source-source prouvées.

Nous proposons d'abord Continuation-Passing C, une extension du langage C pour l'écriture de systèmes concurrents qui offre des threads très légers et unifiés (coopératifs et préemptifs). Les programmes CPC sont transformés par le traducteur CPC pour produire du code à événements séquentialisé efficace, utilisant des threads natifs pour les parties préemptives. Nous définissons et prouvons ensuite la correction de ces transformations, en particulier le lambda lifting et la conversion CPS, pour un langage impératif. Enfin, nous validons la conception et l'implémentation de CPC en le comparant à d'autres bibliothèques de threads et en exhibant notre seeder BitTorrent Hekate. Nous justifions aussi notre choix du lambda lifting en implémentant eCPC, une variante de CPC utilisant les environnements, et en comparant ses performances à celles de CPC.

Biographie :
Gabriel Kerneis, ingénieur Télécom ParisTech et ancien élève de l'ENS Cachan, a effectué un doctorat d'informatique au laboratoire PPS de l'Université Paris Diderot. Durant sa thèse, il s'est attaché à définir, réaliser et montrer la correction de CPC, un langage de programmation pour l'écriture de systèmes concurrents. Il est actuellement chercheur post-doctoral à l'université de Cambridge, où il étudie la sémantique des microprocesseurs dans l'équipe de Peter Sewell.

Posters et démonstrations

Visualisation de données en provenance de capteurs : Vers une visualisation adaptable à l'usage

Ivan Logre, Sébastien Mosser, Anne-Marie Déry, and Michel Riveill

Laboratoire I3S (UMR CNRS-UNS 7271), Université Nice-Sophia Antipolis
{logre, mosser, pinna, riveill}@polytech.unice.fr

1 Motivations

Pour exploiter la masse de données générées par notre environement, les hommes ont besoin de les organiser et de les visualiser sous une forme adaptée à leurs objectifs. Ainsi, plusieurs formats de visualisations sont nécessaires pour permettre l'analyse d'un jeu de données, en fonction de l'expertise et de l'intention de l'analyste [1]. Si l'on considère un coureur s'entraînant régulièrement sur un circuit, la visualisation de sa position GPS seule sur une carte n'a pas d'intérêt. Cependant, en composant cette information avec les données d'un cardio-fréquencemètre et celles en provenance d'un capteur de vitesse, le sportif peut analyser ses performances afin d'optimiser ses entraînements.

Dans ce contexte, la problématique initiale est de fournir un support logiciel aux developpeurs pour composer capteurs et visualisations. De plus, si l'objectif de l'utilisateur n'est plus la performance du sportif mais son suivi médical, la visualisation des mêmes données doit être adaptée à cette tâche. Par exemple, l'interface doit être adaptée afin de visualiser le temps de récupération nécessaire au sportif pour retrouver une fréquence cardiaque (FC) stable après une période d'accélération. En consequence, l'outil doit fournir des mécanismes facilitant l'adaptation de l'application conçue, de manière automatisée [2].

Trois défis sont à relever dans ce contexte pour prendre en compte la corrélation *(i)* entre certains types de capteurs et certains types de visualisation - une position GPS est par exemple fortement corrélée à un affichage sur une carte -, *(ii)* entre certains capteurs, au niveau des instances - dans le scénario ci-dessus, on souhaite composer la vitesse et la FC si elles se réfèrent au même objet d'étude - et *(iii)* entre certaines visualisations, par exemple composer une visualisation par carte avec un gradient de couleur pour étudier l'influence de la vitesse sur la FC.

2 Définition d'un Langage Spécifique au Domaine (DSL)

Nous avons mis en place une solution s'appuyant sur un DSL orienté composant, permettant la modélisation des capteurs déclarés dans SensApp (une plateforme de collecte de données développée dans le cadre des plusieurs projets européens [3]), ainsi que la description des visualisations permettant d'afficher les données associées à ces capteurs. Les concepts de ce DSL ont été capturés avec l'aide d'experts du domaine.

Sur la base de ce DSL, l'utilisateur peut exprimer une composition à un niveau d'abstraction proche de son métier. Dans le Listing 1.1, un capteur d'altitude ainsi qu'une visualisation sous forme d'altimètre sont déclarés (lignes 1 et 6), puis composés par la définition d'un connecteur (ligne 15) au sein d'un composant composite (ligne 12). La mise en œuvre technique du compilateur et des générateurs de codes repose sur la pile logicielle Eclipse/EMF/XText et permet d'atteindre le standard HTML5 comme support de visualisation. Une vidéo montrant un exemple de plateforme de visualisation atteignable est disponible à l'adresse : `http://youtu.be/6CSUQJ5ad1Q#t=2m25s`.

3 Perspectives : Vers une adaptation à l'usage

Le DSL présenté dans cette démonstration propose une solution technique aux défis énoncés en permettant l'association d'un capteur à une visualisation, mais aussi par la déclaration de composants composites en support à la corrélation.

```
sensor BikeAltitude:
  hasForType  Numerical  // Les données sont des nombres
  isEncodedAs SenML  // Utilisant le standard SenML (standardisation IETF)
  isProvidedBy "http://demo.sensapp.org/..." // Ressource SensApp (URL)

widget Altimeter:
  altitude:  // Une visualisation peut prendre plusieurs entrées
    hasForType Numerical
    expectsAsEncoding SenML
  isImplementedBy "gauge.js" // Patron utilisé par le générateur de code

composite MyDashboard: // Assemblage par composition de composants
  myBikeAltitude isA  BikeAltitude
  myAltimeter  isAn Altimeter
  myBikeAltitude.data <-> myAltimeter.altitude // Création d'un connecteur
  // Un composant "capteur" définit implicitement un port "data"
```

Listing 1.1. Example d'utilisation du DSL pour visualiser l'altitude d'un cycliste.

Cependant, l'hypothèse implicite de cet outil est que l'utilisateur détient la connaissance nécessaire à la création de ces associations. Or, en capturant au niveau méta-modèle les propriétés qui permettent d'identifier les associations entre capteur(s) et visualisation(s) corrélés, il est possible de les proposer à l'utilisateur lorsqu'il cherche à composer ces éléments. Pour être utilisable et sensée, cette solution doit s'appuyer sur les besoins de l'utilisateur pour chacune de ses tâches, et proposer un guidage à la création d'un ensemble de visualisations cohérentes par rapport à celles-ci.

On peut ainsi imaginer un outil dédié à des experts du domaine - par exemple le domaine sportif - qui pour un certain profil d'utilisateur - ici des sportifs professionnels et des médecins du sport - et pour chacune des tâches identifiées, proposerait les visualisations adaptées, en présence des bon capteurs. Cette ambition repose sur une capture de la variabilité, à la croisée de la recherche en Ingénierie des Modèles et en Interaction Hommes-Machines.

Remerciements. La mise en œuvre technique du langage a été effectuée en collaboration avec Nohri Graoudi, Stéphane Muller, Thomas Plissonneau et Antoine Pultier.

Références

1. Pinna-Déry, A.M., Gutierrez Rodriguez, C.C., Occello, A. : How a Patient Might Adjust his Cochlear Implant by Using a Smartphone. In : IADIS International Conference e-Health 2012 (e-Health'2012), Lisbon, InderScience (July 2012) 4
2. Mosser, S., Blay-Fornarino, M., Duchien, L. : A Commutative Model Composition Operator to Support Software Adaptation. In : 8th European Conference on Modelling Foundations and Applications (ECMFA'12), Copenhagen, Denmark, Springer LNCS (July 2012) 4–19
3. Mosser, S., Fleurey, F., Morin, B., Chauvel, F., Solberg, A., Goutier, I. : SENSAPP as a Reference Platform to Support Cloud Experiments : From the Internet of Things to the Internet of Services. In : Management of resources and services in Cloud and Sky computing (MICAS), Timisoara, IEEE (September 2012)

Funded under grant ANR-09-SEGI-014

TASCCC - Project and Testing Tool

F. Dadeau[1], K. Cabrera Castillos[1], Y. Ledru[2], L. du Bousquet[2], T. Triki[2], G. Vega[2], S. Taha[3], B. Legeard[1,4], J. Botella[4], B. Chetali[5], J. Bernet[5], D. Rouillard[6]

[1]Femto-ST, [2]LIG, [3]Supélec, [4]Smartesting, [5]Trusted Labs, [6]Serma Technologies

The TASCCC project (2009-2012)

Assistance in the process of a Common Criteria (CC) evaluation by providing:

- a means for developers to generate test cases, focusing on the question of "what to test?" instead of "how to test?"
- a means for evaluators to easily check that the validation was conducted accordingly to the CC norm
- a methodology to produce test reports compliant with CC requirements (format, test description, coverage analysis)

Approach

(1) Informal security requirements are formalized using property patterns.

Dedicated **test property coverage criteria** are then used to:

(2) Evaluate the relevance of a test suite by measuring the coverage of the underlying property automaton.

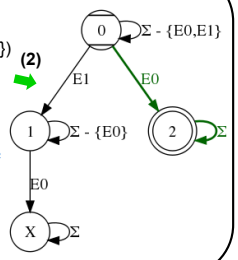
(3) Automatically generate test scenarios by applying dedicated strategies (functional or robustness). Test suites are then generated by animating scenarios on a UML/OCL model using TOBIAS and the Smartesting CertifyIt engine.

"To buy a ticket, the user must be logged on the system."

(1)

```
never isCalled(buyTicket,{@AIM:SUCCESS})
before isCalled(login, {@AIM:SUCCESS})
```

(2)



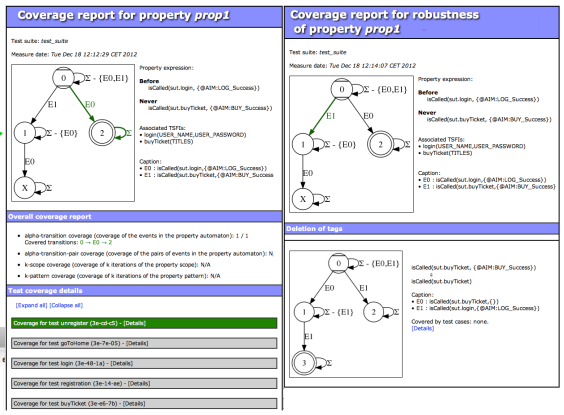
(3)

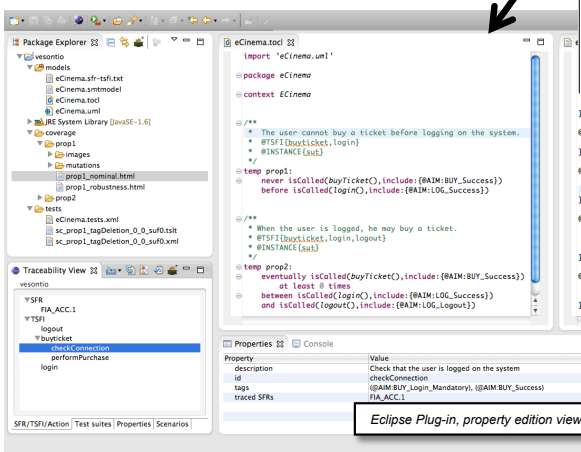
```
sut.buyTicket(TITLE1);
sut.login(REG_USER, REG_PWD);
```

(Try to purchase a ticket, then perform a successful login)

Results of the project

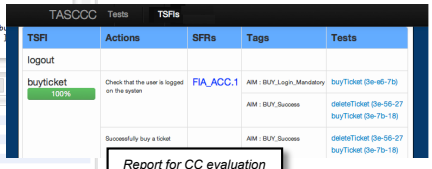
- **Temporal OCL**, a temporal property language
- A **scenario-based testing** approach using UML/OCL models
- Dedicated **test selection criteria**
- An **Eclipse Plug-in** supporting the **whole approach** from property edition to test generation reports for Common Criteria evaluations
- A **successful application** in the context of a CC evaluation of the GlobalPlatform case study





Eclipse Plug-in, property edition view

Flash to see a video demo



Report for CC evaluation

References

Web: http://disc.univ-fcomte.fr/TASCCC

- K. Cabrera, F. Dadeau, J. Julliand and S. Taha. Measuring Test Properties Coverage for evaluating UML/OCL Model-Based Tests. ICTSS'2011.
- T. Triki, Y. Ledru, L. du Bousquet, F. Dadeau, J. Botella, Model-based filtering of combinatorial test suites. FASE'2012.
- F. Dadeau, K. Cabrera, Y. Ledru, T. Triki, G. Vega, S. Taha, J. Botella. Test Generation and Evaluation from High-Level Properties for Common Criteria Evaluations – The TASCCC Testing Tool. ICST'13 Testing Tools track.

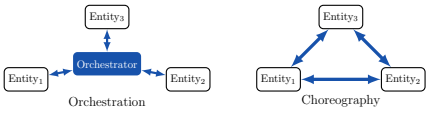
Symbolic Approach for the Verification and the Testing of Service Choreographies

Huu Nghia NGUYEN[1], Pascal POIZAT[2], and Fatiha ZAÏDI[1]

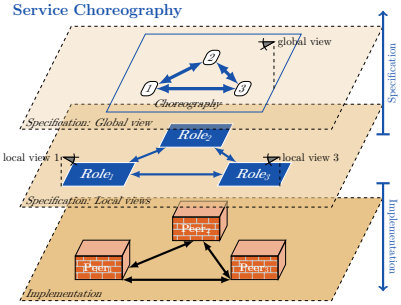
[1] LRI; Univ. Paris-Sud, CNRS, Orsay, France
[2] LIP6; Univ. Paris Ouest Nanterre La Défense, CNRS, La Défense, France
email: huu-nghia.nguyen@lri.fr, pascal.poizat@lip6.fr, fatiha.zaidi@lri.fr

1. Context

Service Composition: Orchestration *vs.* Choreography



Service Choreography

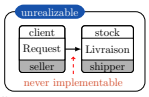


Notation

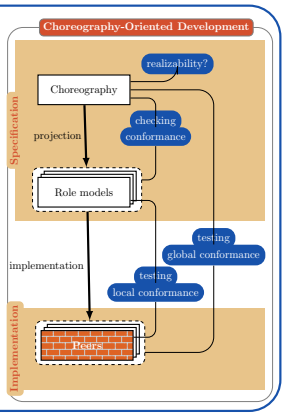
- **Global view:** a `Request` *interaction* between `client` and `seller`
- **Local view of `client`:** *send* `Request` to `seller`
- **Local view of `seller`:** *receive* `Request` from `client`

2. Issues

1. **Modeling**
   - lack or poor data support
   - abstraction ⇒ false negatives
   - enumeration ⇒ state space explosion
2. **Choreography Analysis**
   - **realizability**: is the choreography locally implementable?



3. **Choreography *vs.* Roles Verification**
   - **conformance**: does the composition of the role models conform to choreography?
   - **projection**: how to extract role models from the choreography?
4. **Choreography *vs.* Peers Testing**
   - **local:** is peer behaviour conform to its role?
   - **global:** is the peers' collaboration conform to choreography?



3. Approach

1. **Models:**
   - 1st class interaction ⇒ *interaction-based* choreography rather than interconnection-based choreography
   - data support without state space explosion ⇒ *symbolic semantics* rather than ground semantics
2. **Verification:**
   - possible refinement of peers w.r.t choreography ⇒ weak bisimilar equivalences
3. **Testing:**
   - peers are not controllable ⇒ *(property-oriented) passive testing*

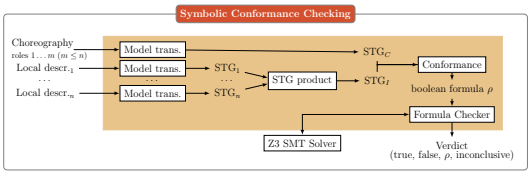
4. Contributions

1. A Language for Choreographies, Roles, and Peers

$L ::= \mathbf{1}$	**inaction**	
$\mid \alpha$	**basic event:**	
	- interaction $o^{[a,b]}.x$	} global model
	- unobservable interaction τ	
	- sending $o^{[a,b]}!x$	} local view
	- reception $o^{[a,b]}?x$	
$\mid x := e$	**assignment:** variable x is substituted by data expression e	
$\mid L_1; L_2$	**sequence:** the events in L_2 are only executed after the ones in L_1	
$\mid L_1 \mid L_2$	**parallel:** there is no order between the events in L_1 and the ones in L_2	
$\mid L_1 + L_2$	**non-deterministic choice:** either L_1 or $L2$ may be executed	
$\mid L_1 \rhd L_2$	**interruption:** the actions in L_1 may be interrupted by an action in L_2	
$\mid [\phi] \triangleright L$	**guard:** L is executed only if the boolean expression ϕ is `true`	
$\mid [\phi] * L$	**loop:** perform L while ϕ is `true`	

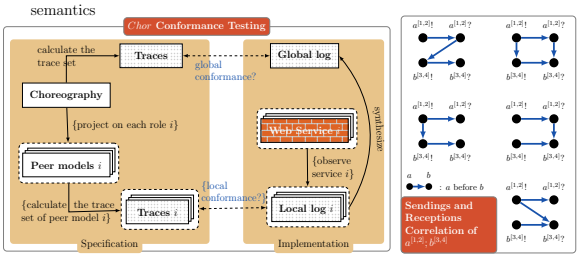
2. Conformance Checking

- symbolic models + extension of a symbolic bisimulation equivalence
- value passing without state space explosion issues
- choreography refinement
- verdict: most general constraint over exchanged information in order to have conformance



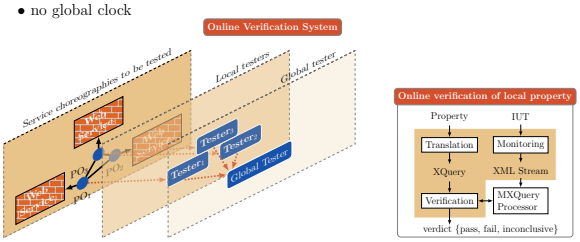
3. Passive Conformance Testing

- trace equivalence conformance relation
- support asynchronous interactions with 5 different possible event correlation semantics



4. Passive Property-Oriented Testing

- properties express critical (positive or negative) behaviors to be tested
- on isolated peers (locally) or sets of peers (globally);
- no global clock



5. Publications

1. H.N. Nguyen, P. Poizat and F. Zaïdi. Online Verification of Value-Passing Choreographies through Property-Oriented Passive Testing. HASE'2012, pp. 106-113, IEEE Computer Society, 2012
2. H.N. Nguyen, P. Poizat and F. Zaïdi. A Symbolic Framework for the Conformance Checking of Value-Passing Choreographies. ICSOC'2012, LNCS 7636:525-532, Springer, 2012
3. H.N. Nguyen, P. Poizat and F. Zaïdi. Passive Conformance Testing of Service Choreographies. SAC'2012, pp. 1528-1535, ACM, 2012

6. Tools

1. **SBBC:** Symbolic Branching Bisimulation for Conformance
2. **Prop-tester:** Online Property-oriented passive testing
3. ***Chor* testing:** conformance testing of *Chor* language
4. **SOAP-capturer:** capture messages between Web services

Freely available at `http://www.lri.fr/~nhnghia/tools`

This work is supported by the ANR Personal Information Management through Internet project (PIMI-ANR-2010-VERS-0014-03)

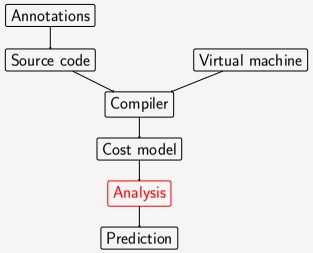
Equipe APR, LIP6 (UMR 7606), Université Pierre et Marie Curie
firstname.lastname@lip6.fr
Jérémie SALVUCCI & Emmanuel CHAILLOUX

MEMORY CONSUMPTION ANALYSIS
FOR APPLICATIVE LANGUAGES

Objectives

- Guarantee a safe upper bound of used memory.
- Consider how **garbage collection** affects this upper bound.
- Reduce the upper bound obtained from existing methods.
- Put a runtime system on an embedded device.

Analysis of a Mini-ML



To create a cost model, we use

- Source code annotation.
- Data representation (data from the virtual machine).
- Compiler code transformation.

The whole process relies on two measures

- An **upper** bound on the amount of heap-allocated memory.
- A **lower** bound on the minimum of recycled memory.

The result of this analysis is the subtraction of these measures.

Targetting a virtual machine

The analysis is performed directly on **bytecode**.

- Explicit stack.
- Garbage Collector root set represented by this stack.
- Tail calls are considered.

We introduce the analysis on program source code (without considering compilation optimizations).

How do we proceed ?

- We use Abstract Interpretation techniques (boolean domain, integer domain (interval)).
- We apply the analysis to monotonic functions.
- We evaluate on both bounds of the interval (depending on function growth).

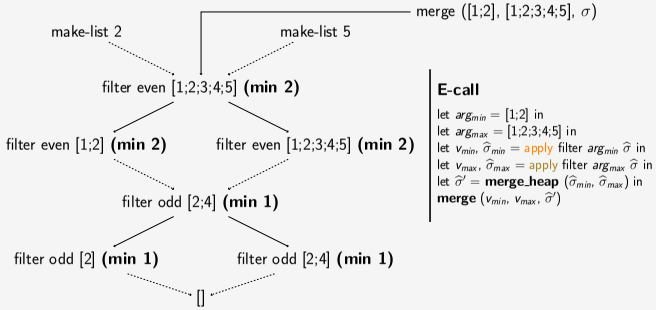
Analysis run on an example

Analysis of the following piece of code

```
let l = if Cond then make-list 2 else make-list 5 (* rule E-if *) in
  filter odd (filter even l) (* rule E-call *)
```

We suppose several things

- make-list n returns a list from 1 to n
- **min n** represents the relevant information to build back the minimal allocated structure.



E-call

let arg_{min} = [1;2] in
let arg_{max} = [1;2;3;4;5] in
let v_{min}, $\widehat{\sigma}_{min}$ = apply filter arg_{min} $\widehat{\sigma}$ in
let v_{max}, $\widehat{\sigma}_{max}$ = apply filter arg_{max} $\widehat{\sigma}$ in
let $\widehat{\sigma}'$ = **merge_heap** ($\widehat{\sigma}_{min}$, $\widehat{\sigma}_{max}$) in
merge (v_{min}, v_{max}, $\widehat{\sigma}'$)

Rules

merge_call (l, v, $\widehat{\sigma}$) :=
  let clss = get (l,$\widehat{\sigma}$) in
  fold (fun acc_v, cls →
    let v, $\widehat{\sigma}'$ = fold (fun acc_v arg →
      let arg_{min} = min arg in
      let arg_{max} = max arg in
      let v_{min}, $\widehat{\sigma}_{min}$ = apply cls arg_{min} $\widehat{\sigma}$ in
      let v_{max}, $\widehat{\sigma}_{max}$ = apply cls arg_{max} $\widehat{\sigma}$ in
      let $\widehat{\sigma}'$ = **merge_heap** ($\widehat{\sigma}_{min}$, $\widehat{\sigma}_{max}$) in
      let v, $\widehat{\sigma}'$ = **merge** (v_{min}, v_{max}, $\widehat{\sigma}'$) in
      merge (acc_v, v, $\widehat{\sigma}'$))
    ϵ
    get (v, $\widehat{\sigma}$)
  in **merge** (acc_v, v, $\widehat{\sigma}'$))
  ϵ
  clss

$$\frac{l \notin dom(\widehat{\sigma}) \qquad \widehat{\sigma}' = \widehat{\sigma}[l \rightarrow (fun\ x \Rightarrow e)[\widehat{\rho}]]}{\widehat{\rho}, \widehat{\sigma} \vdash fun\ x \rightarrow e \rightsquigarrow l} \quad \textbf{(E-fun)}$$

$$\frac{\widehat{\rho}, \widehat{\sigma} \vdash e_0 \rightsquigarrow l, \widehat{\sigma}' \qquad \widehat{\rho}, \widehat{\sigma}' \vdash e_1 \rightsquigarrow v_s, \widehat{\sigma}'' \qquad v, \widehat{\sigma}'' = merge\_call(l, v_s, \widehat{\sigma}'')}{\widehat{\rho}, \widehat{\sigma} \vdash e_0\ e_1 \rightsquigarrow v, \widehat{\sigma}'''} \quad \textbf{(E-call)}$$

$$\frac{\widehat{\rho}, \widehat{\sigma} \vdash e_{cd} \rightsquigarrow \top_b, \widehat{\sigma}' \qquad \widehat{\rho}, \widehat{\sigma}' \vdash e_{cs} \rightsquigarrow v_{cs}, \widehat{\sigma}_{cs} \qquad \widehat{\rho}, \widehat{\sigma}' \vdash e_{alt} \rightsquigarrow v_{alt}, \widehat{\sigma}_{alt}}{\widehat{\rho}, \widehat{\sigma} \vdash if\ e_{cd}\ then\ e_{cs}\ else\ e_{alt} \rightsquigarrow merge(v_{cs}, v_{alt}, merge\_heap(\widehat{\sigma}_{cs}, \widehat{\sigma}_{alt}))} \quad \textbf{(E-if)}$$

Concrete domains	Abstract domains
$z \in Num \equiv \mathbb{Z}$	$\widehat{z} \in \widehat{Num} \equiv \{\bot_n\} \cup \{[a,b] \mid a \in \mathbb{Z},\ b \in \mathbb{Z}\}$
$b \in Bool \equiv \{False, True\}$	$\widehat{b} \in \widehat{Bool} \equiv \{\bot_b, False, True, \top_b\}$
$l \in Loc \equiv \mathbb{N}$	$\widehat{l} \in \widehat{Loc} \equiv \mathbb{N} \cup \{\top_l\}$
$v_s \in V_S \equiv Num + Bool + Loc$	$\widehat{v_s} \in \widehat{V_S} \equiv \widehat{Num} + \widehat{Bool} + \widehat{Loc}$
$\rho : Name \rightarrow V_S$	$\widehat{\rho} : Name \rightarrow \widehat{V_S}$
$dc \in DC \equiv K\ \vec{v_s}$ with $v_s \in V_S$	$\widehat{dc} \in \widehat{DC} \equiv \mathcal{P}(DC)$
$cls \in Cls \equiv Code \times \vec{v_s}$ with $v_s \in V_S$	$\widehat{cls} \in \widehat{Cls} \equiv \mathcal{P}(Cls)$
$v_h \in V_H \equiv DC + Cls$	$\widehat{v_h} \in \widehat{V_H} \equiv \widehat{DC} + \widehat{Cls}$
$\sigma : Loc \rightarrow V_H$	$\widehat{\sigma} : Loc \rightarrow \widehat{V_H}$

Abbreviations

- Loc : Addresses
- DC : Data Constructors
- Cls : Closures
- V_S : Immediate values
- V_H : Allocated values
- ρ : Environment
- σ : Heap

Restrictions

- Analysed programs should be terminating.
- Monotonic functions.
- No side-effects.
- Only linear data structures are allowed.

Current issues

- Notion of **shapes** for nonlinear data structures.
- Adapt the same method to nonlinear data structures is difficult.
- Termination of the analysis.
- Liveness analysis required.
- References are part of a future work.

Related Works

The embounded project - Pedro Vasconcelos' work on sized types

- $Nil :: \forall i < \forall a.List^i a, i = 0 >$
- $Cons :: \forall ij < \forall a.(a, List^i a) \rightarrow List^j a, j = i + 1, i >= 0 >$
- $append :: (List^i a, List^j a) \rightarrow List^k a, k = i + j$

$$\frac{\Gamma_1 \vdash l_2 : List^i a \qquad \dfrac{\Gamma_3 \vdash append(xs, l_2) : List^{k'} a, k' = i' + j' \qquad \Gamma_4 \vdash Cons(x, r) : List^k a, k = 1 + k'}{\Gamma_2 \vdash let\ r = append(xs, l_2)\ in\ Cons(x, r) : List^m a}}{\Gamma_0 \vdash fun\ x\ with\ Nil \rightarrow l_2 | Cons\ (x, xs) \rightarrow let\ r = append(xs, l_2)\ in\ Cons(x, r) : List^n a}$$

RAML - Martin Hofmann's work on automatic amortized analysis

Hypothesis : $append :: (List(A, b_1), List(A, b_2), c) \rightarrow (List(A, b_3), d)$

$$\frac{\Gamma_1, n_1 \vdash l_2 : List(A, a_2), m_1 \qquad \dfrac{\Gamma_2, n_2 \vdash append\ xs\ l_2 : List(A, a_4), m_2}{\Gamma_3, n_3 \vdash Cons(x, r), n_3 : List(A, m_3)}}{\Gamma_0 \vdash fun\ l_1\ with\ Nil \rightarrow l_2 | Cons(x, xs) \rightarrow let\ r = append\ xs\ l_2\ in\ Cons(x, r))}$$

The analysis consists in finding the coefficients of the potential function which is a linear combination of the function inputs.

References

P. Hudak.
A Semantic Model of Reference Counting and its Abstraction.
In *Abstract Interpretation of Declarative Languages*, pages 45–62. Ellis Horwood, 1987.

Steffen Jost.
Automated Amortised Analysis.
PhD thesis, Faculty of Mathematics, Computer Science and Statistics, LMU Munich, Germany, September 2010.

Pedro Vasconcelos.
Space Cost Analysis Using Sized Types.
PhD thesis, School of Computer Science, University of St Andrews, Scotland, 2008.

Cinquièmes journées nationales du GDR GPL – 3 au 5 avril 2013

Optimiser et Répartir ses Applications Mobiles avec Macchiato

Nicolas Petitprez, Romain Rouvoy et Laurence Duchien

Inria Lille – Nord Europe,
LIFL - CNRS UMR 8022,
Université de Lille 1, France
{nicolas.petitprez,romain.rouvoy,laurence.duchien}@inria.fr

1 Introduction

De nos jours, les utilisateurs de terminaux mobiles souhaitent des applications de plus en plus personnalisées et riches en fonctionnalités et consommant de plus en plus de données. Ces nouveaux usages se traduisent par une surconsommation des ressources du périphérique (par exemple, utilisation du processeur et de la connexion réseau). Sur un périphérique mobile, ces ressources sont limitées, principalement par la capacité de la batterie et les limitations technologiques et contractuelles du réseau. Il faut donc minimiser l'utilisation des ressources du périphérique, tout en maximisant les performances de l'application.

Dans un contexte de mobilité, l'ensemble des traitements peut être effectué sur le téléphone mobile, mais cela sollicite fortement les ressources du périphérique. Il est possible de déléguer les traitements à un serveur distant, mais, dans ce cas, les performances sont liées à celles du serveur et à la qualité de la connexion réseau. En cas de coupure de connexion, l'application cesse totalement de fonctionner. Une solution est donc d'envisager de répartir dynamiquement les traitements entre le périphérique et des serveurs distants tout en garantissant les performances et en étant tolérant à l'absence de connectivité réseau.

Certains travaux présentent les techniques de répartition d'une application mobile entre un périphérique mobile et un serveur [2,3]. Il est ainsi possible d'améliorer les temps de réponse d'une application et de réduire sa consommation de ressources [4]. Malheureusement, les solutions de répartition des traitements mobiles existantes sont coûteuses à mettre en place, complexes à utiliser, et n'offrent pas de support pour permettre l'adaptation des applications à l'environnement d'exécution. De plus, dans les applications mobiles, cet environnement d'exécution évolue en permanence : niveau de charge de la batterie, présence de la géolocalisation, qualité de la connexion réseau. Il est important d'offrir un support aux développeurs pour pouvoir facilement adapter les traitements à cet environnement et les répartir.

2 La Plate-forme Macchiato

Nous présentons dans cette démonstration la plate-forme Macchiato [1], développée pour construire facilement des applications mobiles performantes, adaptables et pouvant être réparties tout en étant économes en ressources. Macchiato propose aux développeurs un langage embarqué qui permet de définir des acteurs répartis exploitant les standards du Web. Les acteurs sont des entités autonomes qui communiquent par envoi de message. Ce modèle asynchrone permet de répartir efficacement les traitements entre différentes machines. Notre plate-forme présente des propriétés importantes qui permettent de répondre aux problématiques énoncées précédemment. Dans la suite, nous détaillons ces propriétés.

Cette plate-forme est **agnostique** vis à vis de l'environnement d'exécution. Les applications développées s'exécutent sans modification sur tous les environnements que la plate-forme supporte. L'utilisation d'un langage de script courant permet de cibler un nombre important d'architectures

1. Projet Macchiato : http://www.macchiato.fr/

d'exécution. Une abstraction de l'environnement permet d'avoir un code identique quel que soit l'environnement d'exécution. Il est actuellement possible d'exécuter des acteurs Macchiato dans les navigateurs Web, dans les applications mobiles *Android* et sur les serveurs d'application *vert.x* [2].

Pour pouvoir s'adapter facilement aux besoins et à l'environnement, la plate-forme est **réflexive**. Elle permet d'introspecter l'architecture de l'application et de reconfigurer son fonctionnement par des opérations d'ajout, de modification ou de suppression d'acteurs.

Pour répondre aux problématiques de performance et d'économie de ressources, la plate-forme est **auto-optimisable**. La figure 1 présente le système d'optimisation de la plate-forme Macchiato. Lors de l'exécution, la plate-forme collecte des données sur le fonctionnement de l'application, comme le nombre et la taille des messages échangés, l'utilisation de la connexion réseau, etc. Les informations collectées permettent de modéliser le problème d'optimisation comme un problème pseudo-booléen [1]. L'utilisation d'un solveur permet alors de trouver la meilleure répartition de l'application en fonction de l'objectif retenu. Il est, par exemple, possible de limiter la quantité de données qui transite par la connexion réseau du périphérique.

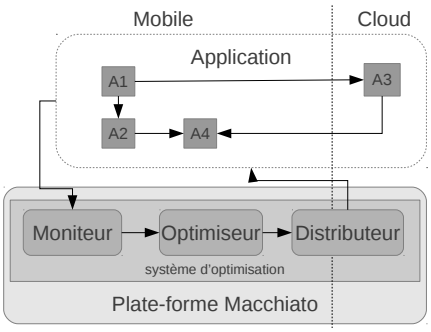


Figure 1. Système d'optimisation de la plate-forme Macchiato

Dans un contexte mobile, il est fréquent d'avoir des problèmes de qualité de connexion. La plate-forme se doit donc d'être **robuste**. Le modèle de l'application ainsi qu'une copie des acteurs déployés à distance sont conservés sur le périphérique du client. En cas de défaillance d'un serveur distant ou de la connectivité du périphérique, la plate-forme redirigera automatiquement les messages à destination du serveur inaccessible vers les acteurs fonctionnant sur le périphérique du client.

Dans notre démonstration, nous utiliserons le langage embarqué de la plate-forme Macchiato pour construire une application mobile. Nous montrerons ensuite comment la plate-forme surveille l'exécution de l'application, planifie son optimisation, et la répartit entre l'environnement du client et un serveur d'application.

Références

1. Daniel Le Berre and Anne Parrain. The Sat4j library, release 2.2 system description. *Journal on Satisfiability, Boolean Modeling and Computation*, 7(2010) :59–64, 2010.
2. Ioana Giurgiu, Oriana Riva, and Gustavo Alonso. Dynamic software deployment from clouds to mobile devices. In *Middleware*, pages 394–414, 2012.
3. Roelof Kemp, Nicholas Palmer, Thilo Kielmann, and Henri Bal. Cuckoo : a computation offloading framework for smartphones. *Mobile Computing, Applications, and Services*, pages 59–79, 2012.
4. K Kumar and YH Lu. Cloud computing for mobile users : Can offloading computation save energy ? *IEEE Computer*, (April) :51–56, 2010.

2. vert.x : `http://vertx.io`

PROGRAMMING EMBEDDED SYSTEMS WITH EVENTS: CASE STUDIES

Truong-Giang Le, Dmitriy Fedosov, Olivier Hermant, Matthieu Manceny, Renaud Pawlak, and Renaud Rioboo

Contact: le-truong.giang@isep.fr
LISITE-ISEP, 28 Rue Notre-Dame des Champs, 75006, Paris, France

INTRODUCTION

- Many embedded systems are event-driven.
  E.g.: data collecting and processing systems, monitoring and controlling systems, automatic selling machines, robotic systems.
- Programming language support for events is still limited.
  - ✓ Events are not defined intuitively and straightforwardly.
  - ✓ Do not take advantage of multithreading.
  - ✓ No dynamic (runtime) behavior adaptation.

OUR APPROACH

- Propose an event-based programming language called INI:
  - ✓ Programmers may use built-in events or write user-defined events in Java/C-C++.
  - ✓ Events run in parallel either asynchronously or synchronously.
  - ✓ Events can be stopped/restarted or reconfigured at runtime to change their behavior.

Syntax:

```
$(<List of synchronized events>) id:@eventKind[<Input parameters>](<Output parameters>) {<Action>}
```

CASE STUDIES

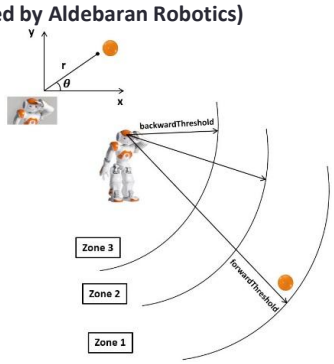
A ball tracking program running on the humanoid robot Nao (manufactured by Aldebaran Robotics)

```
function main() {
  @init() {
    forwardThreshold=0.5
    backwardThreshold=0.3
    stepFrequency=0.0
    defaultStepFrequency=1.0
    targetTheta=0.0
    robotPosition=[0.0,0.0,0.0]
    stepX=0.0
    needAdjustDirection=false
    i=0
  }
  $(e) b:@ballDetection[robotIP="local",port=9559,checkingTime=1000]
   (ballPosition) {
    //Compute necessary parameters, and return in an array
    parameters=process_position(ballPosition, forwardThreshold,
     backwardThreshold)
    targetTheta=parameters[0]
    robotPosition=parameters[1]
    stepX=parameters[2]
    needAdjustDirection=true
    stepFrequency=defaultStepFrequency
    i=0
  }
  $(b,e) e:@every[time=200]() {
    //Control the robot to go one step if the ball is detected
    needAdjustDirection=reach_to_target(...)
    i++
    case {
      //Reset parameters after three consecutive walking steps
      i>3 {
        stepX=0.0
        targetTheta=0.0
        stepFrequency=0.0
      }
    }
  }
}
```

The built-in event @init is invoked when a function starts. Normally, it is used to initialize necessary variables.

The user-defined event @ballDetection is invoked when a ball is detected. Inside this event, we compute parameters needed for controlling Nao's movement.

The built-in event @every is invoked to control the robot to move.



The ball is in zone 1: Nao will go forward to reach the ball.
The ball is in zone 2: Nao does not move since its current position is appropriate to observe the ball.
The ball is in zone 3: Nao will go backward to avoid a possible collision since it assumes that the ball is too close and also is moving.

Results:

- Nao detects the ball in the space and then walks towards it.
- In case that the ball is moved to another place, Nao adjusts the direction and velocity in order to follow it.

Another case study:

- We also apply INI in an industrial M2M gateway, which captures and transmits data through the network. The data collection frequency can be dynamically modified at runtime to adapt to the battery level.

Reference:
https://sites.google.com/site/inilanguage

Project: MCUBE

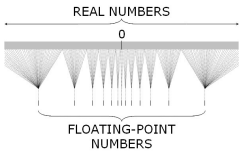
Research partners and sponsors: isep, Cédric le cnam, SYSTEMATIC PARIS REGION SYSTEMS & ICT CLUSTER, l'Europe s'engage en Île-de-France

Elimination des racines et divisions pour du code embarqué

Pierre NERON
Inria / École polytechnique

1 . Contexte

- Code embarqué critique :
  - système ACCoRD pour l'aéronautique (NASA Langley)
  - operateur conditionnel (if then else)
  - programmes sans boucles
  - pas d'allocation dynamique de mémoire
- Arithmétique réelle $+, -, \times, /, \sqrt{}$
- Représentation finie des réels en informatique :



$\Longrightarrow \sqrt{2} \times \sqrt{2} > 2$

- $\sqrt{}$ et $/$ créent des suites infinies :
  $\sqrt{2} = 1.41421356237...$ $\quad 1/7 = 0.14285714285...$
- Arithmétique exacte avec $+, -, \times$:
  - entiers dynamiques
  - taille max par analyse statique

2 . Spécification

Étant donné qu'il est possible de calculer exactement avec $+, -, \times$, le but est de définir une **transformation de programmes** qui :

- élimine les racines et les divisions
- préserve la sémantique lorsqu'il n'y a pas d'échec

Si on ne peut pas toujours éliminer ces opérations (le programme $\sqrt{2}$ retournera toujours une valeur approchée), on peut en revanche les éliminer de valeurs booléennes et ainsi protéger le graphe de contrôle du programme des erreurs d'arrondis.

3 . Langage

Prog :=	Constant	| Var
	| fst Prog	| snd Prog
	| uop Prog	| Prog op Prog
	| (Prog, Prog)	| let Var = Prog in Prog
	| if Prog then Prog else Prog	

avec : Constant $\subset \mathbb{R} \cup \{True,\ False\}$
op $\in \{+, \times, /, =, \neq, >, \geq, <, \leq, \wedge, \vee\}$
uop $\in \{\sqrt{}, -, \neg\}$

4 . Expressions booléennes

Soit $E_1\ \mathcal{R}\ E_2$ une comparaison, on élimine racines et divisions en appliquant les transformation suivantes qui définissent la fonction `elim_bool` :

- Mettre les divisions en tête :

$$E_1\ \mathcal{R}\ E_2 \longrightarrow \frac{A}{B}\ \mathcal{R}\ \frac{C}{D}$$

- Eliminer les divisions de tête :

$$\frac{A}{B}\ \mathcal{R}\ \frac{C}{D} \longrightarrow A.B.D^2\ \mathcal{R}\ C.D.B^2$$

- Choisir une racine et factoriser :

$$A.B.D^2\ \mathcal{R}\ C.D.B^2 \longrightarrow P.\sqrt{Q} + R\ \mathcal{R}\ 0$$

- Eliminer la racine choisie : $P.\sqrt{Q} + R\ \mathcal{R}\ 0 \longrightarrow$

$$(P\ \mathcal{R}\ 0 \wedge R\ \mathcal{R}\ 0) \vee (P \geq 0 \wedge P^2.Q - R^2\ \mathcal{R}\ 0) \vee (R \geq 0 \wedge 0\ \mathcal{R}\ P^2.Q - R^2)$$

- Tant qu'il y a des racines, recommencer

5 . Définitions de variables

Afin d'éviter que ces expressions booléenes ne dependent de racines ou de divisions indirectement, on élimine également ces opérations des définitions de variable en utilisant un *inlining* partiel :

$\diamond$ let x $= a.b + \sqrt{(c+d)/f}$ in P $\longrightarrow$
let $(x_1,x_2,x_3) = (a.b, c+d, e)$ in P[x := $x_1 + \sqrt{x_2/x_3}$]

- Nommer les sous expressions qui ne contiennent ni racine ni division
- *Inliner* le contexte qui les contient

6 . Définitions avec conditionnelles

Cette notion d'*inlining* partiel peut s'étendre à des définitions de variables qui contiennent des tests :

$\diamond$ let x = if F then a/b else $c + \sqrt{d}$ in P

Le but est alors de trouver une representation commune à toutes les expressions qui correspondent aux différents cas et d'*inliner* cette expression :

$\diamond$ let (x_1,x_2,x_3) = if F then $(a,b,0)$ else $(c,1,d)$ in P[x := $\frac{x_1+\sqrt{x_3}}{x_2}$]

Cette représentation commune provient d'une **anti-unification** avec contraintes des expressions correspondant aux différents cas des tests.

Soient $e_1, ..., e_n$, un anti-unificateur de ces termes est un terme t tel que :

$$\forall i \in [|1, ..., n|],\ \exists \sigma_i \in \mathcal{P}erm(\mathrm{Var}),\ t.\sigma_i = e_i$$

Avec la contrainte que σ_i ne contient ni racine ni division.

Cette anti-unification nous permet donc de definir une fonction `elim_let(x,p1,p2)` qui renvoit x', p1', p2' tels que :

$$\text{let x} = \text{p1 in p2} \overset{sem}{==} \text{let x'} = \text{p1' in p2'}$$

où p1' ne contient pas de racine.

7 . Transformation complète

La transformation est donnée par la fonction récursive `Elim(p)` :

- si p est une expression booléene, retourner `elim_bool(p)`
- si p est une expression arithmétique, retourner p
- si p = let x = p1 in p2 :
  - p1r `:= Elim(p1)`
  - x',p1',p2' `:= elim_let(x,p1r,p2)`
  - retourner let x' = p1' in `Elim(p2')`
- si p = if F then p1 else p2
  retourner if `Elim(F)` then `Elim(p1)` else `Elim(p2)`

8 . Conclusion

Nous avons donc conçu une transformation de programme qui permet d'éliminer les racines et les divisions de tous les booléens d'un programme. Cette transformation est implantée en OCamL.

De plus nous avons :

- une spécification et la preuve de correction en PVS
- transposé cette spécification en une tactique réflexive qui permet de transformer automatiquement des buts dans PVS

9 . Référence

P. Neron. A formal proof of square root and division elimination in embedded programs. In C. Hawblitzel and D. Miller, editors, CPP, volume 7679 of *Lecture Notes in Computer Science*, pages 256–272, 2012.

http://www.lix.polytechnique.fr/~neron/

Modèle de Défaillances Sûres pour des Applications Ferroviaires Critiques

Développement à Base de Composants

Marc Sango, Laurence Duchien, Christophe Gransart
Univ Lille Nord de France - IFSTTAR - INRIA - Univ Lille 1 - LIFL, UMR CNRS 8022
marc.sango@ifsttar.fr, laurence.duchien@inria.fr, christophe.gransart@ifsttar.fr

Contexte

Les applications critiques sont conçues de telle façon qu'elles continuent à fonctionner en conformité avec leurs spécifications de sûreté malgré des fautes résiduelles. Cette technique de conception est connue sous le nom de la tolérance aux fautes. La tolérance aux fautes comprend plusieurs modes de défaillances qui sont définis par leurs sémantiques d'interprétation.

Dans certains domaines critiques comme dans le ferroviaire, **les applications restent en grande majorité développées manuellement et les différentes préoccupations extra fonctionnelles, telle que la tolérance aux fautes, sont intégrées dans le code fonctionnel** sous forme d'algorithmes, de protocoles, et ou encore d'exceptions.

Construites ainsi, les architectures logicielles de ces applications sont complexes à faire évoluer et encore plus difficiles à adapter à l'exécution.

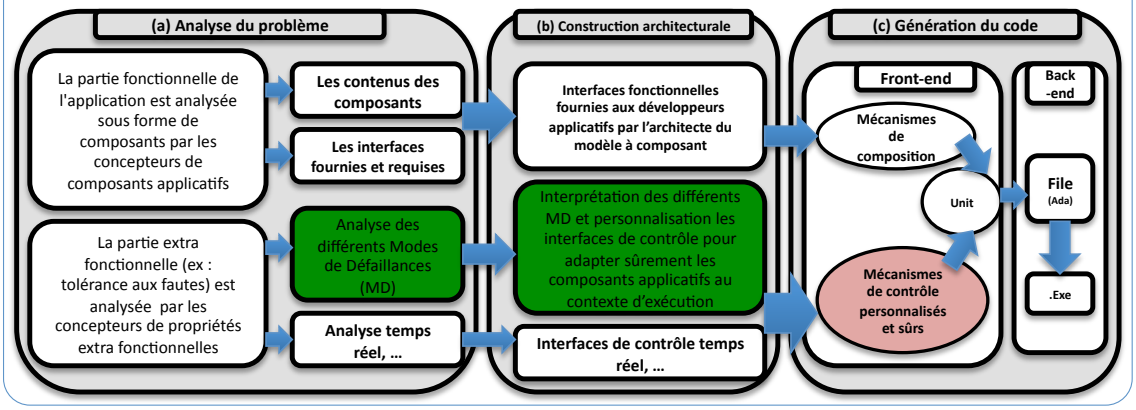
Objectif, Approche et Proposition

Objectif : Séparation de préoccupations dans tout le cycle de vie logiciel pour préserver les propriétés de sûreté de la conception à l'exécution.

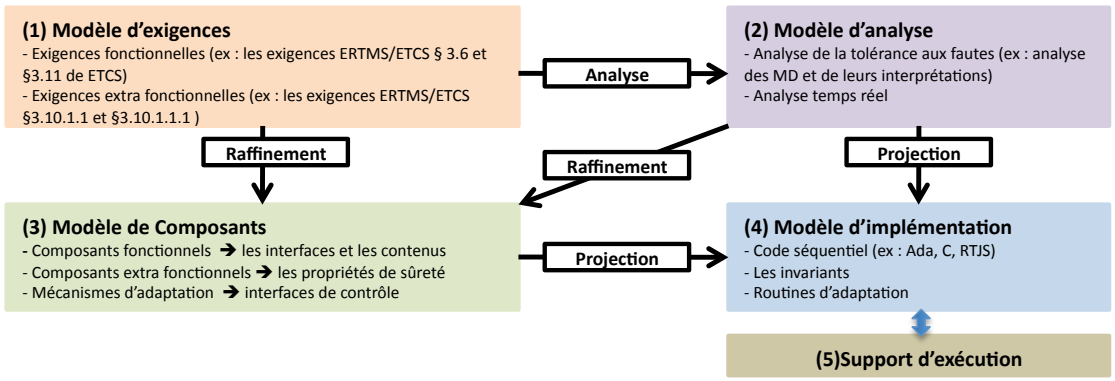
Approche : Séparation des préoccupations des fonctionnalités classiques des fonctionnalités de tolérances aux fautes, i.e., définition des différents modes de défaillance qui impactent la sûreté de fonctionnement et interprétation de la conséquence de ces défaillances.

Proposition : Utilisation d'une approche à base de composants qui mettra en œuvre la séparation des préoccupations de la conception à l'exécution.

Approche générale



Méthodologie de développement



Plan de travail

➢Définition d'un modèle de composants comportant le modèle de défaillances amélioré et qui permettra de réaliser la séparation de préoccupations jusqu'à l'exécution
➢Implémentation du modèle dans un langage recommandé dans le ferroviaire
➢Expérimentation sur un exemple ERTMS/ETCS

Conclusion

➢Proposition d'un modèle de modes de défaillances et de leurs interprétations
➢Evaluation du système ERTMS/ETCS & Norme Ferroviaire EN 50128
➢Evaluation de Fractal

Références

➢ A. Bondavalli and L. Simoncini. Failure classification with respect to detection. In Distributed Computing Systems, 1990. Proceedings., Second IEEE Workshop on Future Trends of, pages 47 –53, sep-2 oct 1990

➢M. Stoicescu, J.-C. Fabre, and M. Roy. From design for adaptation to component-based resilient computing. In PRDC, pages 1–10, 2012.

Université Lille1 Sciences et Technologies
Laboratoire d'Informatique Fondamentale de Lille
informatics mathematics Inria

Automated Runtime Software Repair

Improving resilience in presence of exceptions

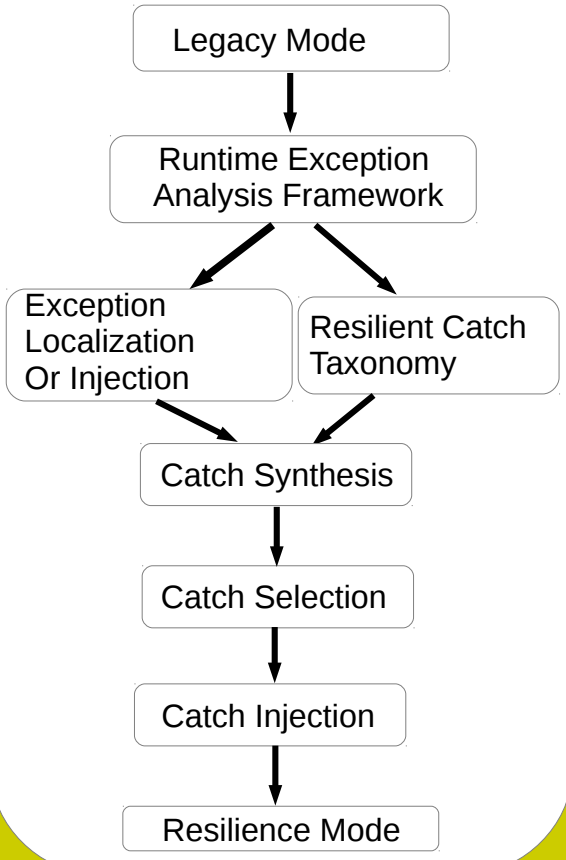
Benoit Cornu, Martin Monperrus

Context

The server encountered an internal error ...

> Observation: m*any applications crash due to unexpected exceptions*
> Goal: *catch those exceptions in an automated manner to prevent crashing the application*
> Techniques: code transformation, empirical analysis, code synthesis, data-mining

Thesis Overview



Runtime Exception Analysis

Applicative Goals
-analyze the exception behavior specification
-isolate the catch blocks needed to perform an action
-isolate the catch blocks which are not (or cannot be) used

Technical Goals
-detection of exception sources and their context
-capture of catch block execution and their interplay
-analyze the relationship between causes and treatments

The Spoon library is used to instrument source code

```
try{
     Framework.learn("try-start");
     //developper code
     Framework.learn("try-end");
}catch(RuntimeException re){
     Framework.learn("catch-start", re);
     //developper code
     Framework.learn("catch-end");
}
```

Results

Run org.apache.commons.lang Test Suite skipping the catch treatments: (2051 tests)
4 failures --> bad assert result
125 errors --> unexpected Exception
1 crash --> non-ending required Thread

Number of used catch blocks / thrown exceptions to boot
Vuze: 24 catch – 413 exceptions
Jabref : 5 catch – 50 exceptions
Bluej : 0 catch – 1 exceptions

Ongoing Work

Exception injection of appropriate types at relevant place.
-select the place according to execution traces
-for a given place choose the type according to the static and dynamic context
-minimize artificiality / maximize learning

Evaluation
-How many "manually" fixed exception bugs from bug repositories can be automatically repaired by the framework?
-How much can the framework avoid crashing from fault injection?

Dedukti : un vérificateur de preuves universel

Ali Assaf[1,2], Raphaël Cauderlier[1,3], and Ronan Saillard[1,4]

[1]INRIA Paris-Rocquencourt
[2]École Polytechnique
[3]ENS Cachan
[4]MINES ParisTech

1 Introduction

Dedukti est un vérificateur de types pour le $\lambda\Pi$-calcul modulo [3]. Ce formalisme, alliant types dépendants et réécriture, permet d'exprimer et de vérifier les preuves de nombreux systèmes logiques [4]. Nous proposons de l'utiliser comme vérificateur de preuves universel. Holide [1], Coqine [2] et Focalide sont des outils auxiliaires qui traduisent respectivement les preuves de HOL, Coq et FoCaLize vers Dedukti. Cette démonstration est présentée en parallèle avec un poster sur le même sujet.

2 Description des outils

2.1 Dedukti

Dedukti est un vérificateur de types pour le $\lambda\Pi$-calcul modulo. Contrairement à la plupart des outils de ce genre, Dedukti implémente une architecture de génération de code : le fichier source est d'abord traduit vers un programme Lua qui est ensuite exécuté pour obtenir le résultat. Cette architecture a de nombreux avantages. En effet elle permet de profiter gratuitement des fonctionnalités calculatoires de Lua, évitant ainsi de devoir réimplémenter certains algorithmes tels que la substitution ou la normalisation des termes. De plus l'utilisation du compilateur *just-in-time* LuaJIT comme *back-end* permet d'avoir des performances optimales, quelque soit la quantité de calcul dans les preuves vérifiées.

2.2 Holide

HOL est une famille d'assistants de preuve basés sur la théorie des types simples de Church. Cette famille inclue HOL Light, HOL4, et ProofPower. Un point commun de ces systèmes est qu'ils ne gardent pas les preuves de leurs théorèmes en mémoire, mais peuvent imprimer une trace de leurs dérivations dans un format appelé OpenTheory. Ce format consiste en une suite de commandes à une machine virtuelle qui décrivent comment reconstituer la preuve.

Holide traduit les preuves de HOL écrites dans le format OpenTheory. Il consiste en une implémentation de la machine virtuelle OpenTheory qui construit en mémoire les termes de preuves correspondants. Ces termes de preuves sont ensuite traduits selon un encodage de la théorie des types simples dans Dedukti.

2.3 Coqine

Coq est un assistant de preuve basé sur le Calcul des Constructions Inductives (CIC), une extension du Calcul des Constructions (COC) avec des types inductifs. Un encodage du COC a été présenté dans [4] ; il a été adapté dans Coqine pour gérer les types inductifs.

Coqine est implémenté comme une extension du noyau de vérification de Coq. Les fichiers de Coq (`.v`) sont d'abord compilés par Coq pour produire des fichiers binaires (`.vo`). Ces fichiers sont ensuite lus et traduits par Coqine selon un encodage du CIC dans Dedukti.

2.4 Focalide

FoCaLize est un environnement de programmation permettant le développement de programmes certifiés. Il est basé sur un langage fonctionnel avec des traits orientés objets. Les fichiers FoCaLize sont traduits par le compilateur, d'une part en programmes OCaml, d'autre part en certificats pour l'assistant de preuve Coq.

Focalide est une extension de ce compilateur ajoutant une troisième sortie produisant un fichier Dedukti.

3 Description de la démonstration

L'exposé consistera en une démonstration de l'utilisation des différents outils de traduction (Holide, Coqine et Focalide), ainsi que la vérification, à l'aide de Dedukti, des preuves produites.

Dedukti, Holide, Coqine et Focalide sont disponibles en ligne sur le site web de Dedukti : `https://www.rocq.inria.fr/deducteam/Dedukti/`.

Références

[1] A. Assaf and G. Burel. Translating hol to dedukti. Submitted to RTA2013.

[2] M. Boespflug and G. Burel. CoqInE : translating the calculus of inductive constructions into the lambda-pi-calculus modulo. In *Proof Exchange for Theorem Proving—Second International Workshop, PxTP 2012*, page 44, 2012.

[3] Mathieu Boespflug, Quentin Carbonneaux, and Olivier Hermant. The lambda-pi-calculus modulo as a universal proof language. In *Proof Exchange for Theorem Proving - Second International Workshop, PxTP 2012*, volume 878 of *CEUR Workshop Proceedings*, page 28, Manchester, UK, June 2012.

[4] Denis Cousineau and Gilles Dowek. Embedding pure type systems in the lambda-pi-calculus modulo. In Simona Ronchi Della Rocca, editor, *Typed Lambda Calculi and Applications*, number 4583 in Lecture Notes in Computer Science, pages 102–117. Springer Berlin Heidelberg, January 2007.

Dedukti : un vérificateur de preuves universel

Ali Assaf, Raphaël Cauderlier et Ronan Saillard

DEDUCTEAM (INRIA) - MINES ParisTech

ali.assaf@inria.fr raphael.cauderlier@inria.fr ronan.saillard@inria.fr

Introduction

Dedukti est un vérificateur de types pour le $\lambda\Pi$-calcul modulo, un formalisme alliant types dépendants et réécriture qui permet d'exprimer et de vérifier les preuves de nombreux systèmes logiques.
Nous proposons d'utiliser Dedukti comme un vérificateur de preuves universel en traduisant *HOL*, *Coq* et *FoCaLize* vers Dedukti.

HOL

```
# let transitivity =
   EQ_MP
    (MK_COMB (REFL `(=) x : A -> bool`,
      ASSUME `y : A = z`))
    (ASSUME `x : A = y`);;

val transitivity :
  thm = x = y, y = z |- x = z
```

Holide

Coq

```
Theorem transitivity :
  forall (A : Type) (x y z : A),
    x = y -> y = z -> x = z.
Proof.
  intros A x y z H1 H2.
  induction H1.
  exact H2.
Qed.
```

Coqine

FoCaLize

```
species Setoid =
  signature (=): Self -> Self -> bool;
  property transitivity :
    all x y z : Self,
      x = y -> y = z -> x = z;
end;;
```

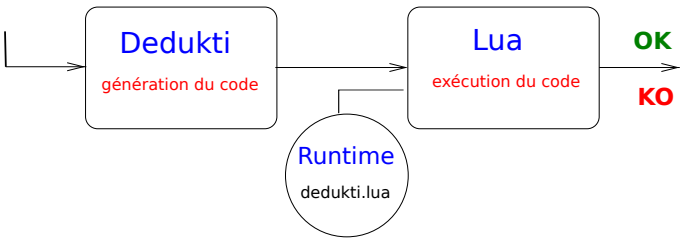
Focalide

Dedukti

```
A : Type.
Nat: Type.
Z : Nat.
S : Nat -> Nat.

plus: Nat -> Nat -> Nat.
[m:Nat] plus Z m --> m
[n:Nat,m:Nat] plus (S n) m --> plus n (S m).

Listn : Nat -> Type.
nil : Listn Z.
cons : n:Nat -> A -> Listn n -> Listn (S n).

append: n:Nat -> Listn n -> m:Nat -> Listn m -> Listn (plus m n).
[n:Nat,l1:Listn n] append n l1 Z nil --> l1
[n:Nat,l1:Listn n,m:Nat,l2:Listn m,a:A]
    append n l1 (S m) (cons {m} a l2) --> append (S n) (cons n a l1) m l2.
```

Définition de la concaténation de listes de taille n en Dedukti.

- Un vérificateur de preuves basé sur le $\lambda\Pi$-calcul modulo (voir encadré).
- Une architecture originale : le fichier source est compilé vers un langage cible (*Lua*) qui est ensuite exécuté pour obtenir le résultat.
- De la normalisation par évaluation : la normalisation est assurée par l'exécution dans le langage cible.
- Une compilation *just-in-time* (*JIT*) : l'utilisation d'un compilateur *JIT* comme *back-end* assure des performances optimales quelque soit la quantité de calcul dans les preuves vérifiées.
- Une vérification de type sans contexte : le contexte est remplacé par des annotations de type.
- Un algorithme bi-directionnel : le système alterne entre des phases de vérification et d'inférence de type, guidé par la structure du terme.



Dedukti est un générateur de code.

Le $\lambda\Pi$-calcul modulo

- Le $\lambda\Pi$-calcul est un λ-calcul avec des types dépendants qui permet d'exprimer les preuves de la logique minimale des prédicats à travers la correspondance de Curry-DeBruijn-Howard.
- Le $\lambda\Pi$-calcul modulo est une extension du $\lambda\Pi$-calcul qui intègre une notion de convertibilité élargie :

$$\frac{\Gamma \vdash t : A \quad \Gamma \vdash B : s \quad A \equiv_{\beta\mathcal{R}} B}{\Gamma \vdash t : B}(\text{Conv})$$

où $\mathcal{R}$ est la congruence générée par un système de réécriture bien typé arbitraire.

Vers l'interopérabilité

- Les systèmes de preuves actuels souffrent d'un manque d'interopérabilité. Il est difficile de réutiliser une théorie d'un système dans un autre sans refaire toutes les preuves.
- La traduction de ces différents systèmes dans un formalisme commun permettra de combiner leurs preuves pour construire des théories plus larges.
- Des traductions d'autres systèmes, tels que *PVS*, *Matita* ou encore *Atelier B*, sont à l'étude.

Dedukti, Holide, Coqine et Focalide sont disponibles sur le site web de Dedukti : `https://www.rocq.inria.fr/deducteam/dedukti/`

femto-st SCIENCES & TECHNOLOGIES

Praspel, a Specification Language and Testing Framework for PHP

Praspel Language

- Praspel = PHP Realistic Annotation and Specification Language
- Contract-based testing language expressing invariants, pre- and post-conditions
- Embedded as annotations in PHP classes
- Used to associate realistic domains to data (class attributes or method parameters)

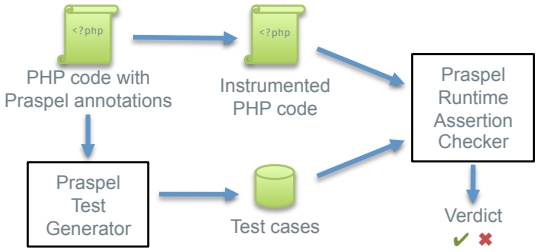
Praspel Testing Framework

- Standard library including more than 30 realistic domains is available
- Algorithms:
  - Random Generation [EDGBO11]
  - Grammar-Based Testing for Strings [EDBG12]
  - Constraint Solver for Arrays [EGB13]
- Framework:
  - Integrated and supported by the Hoa community
  - Free and open-source

Realistic Domains

- Used to specify data domains relevant for specific application contexts
- Tailored for dynamic and weakly typed languages
- Defined by two properties:
  1. predicability: checks if a value belongs to a realistic domain
  2. samplability: generates a value that belongs to a realistic domain
- Implemented in PHP as classes

Test Generation and Execution




```
class C {

   /** @invariant I_1 and ... and I_h */

   /**
    * @requires  R_1 and ... and R_n
    * @ensures   E_1 and ... and E_j;
    * @throwable T_1, ..., T_t;
    * @behavior α {
    *    @requires  A_1 and ... and A_k;
    *    @ensures   E_{j+1} and ... and E_m;
    *    @throwable T_{t+1}, ..., T_l;
    * }
    */
   function foo ( $x_1... ) { body }
   ...
}
```

```
length: 0..5 or 10 and
arr   : array([to string('a', 'e', 1)], length) and
arr[0]: 'b' or 'd' and
\pred ($arr[0] < $arr[1]) and
user  : EmailAddress()
```

```
class EmailAddress extends String {

  public function predicate ( $q ) {

    return    parent::predicate($q)
           && 0 !== preg_match(…, $q);
  }

  public function sample ( Sampler $s ) {

    // Generate a valid email address.
    return $data = …;
  }
}
```

References

[EDGBO11] Ivan Enderlin, Frédéric Dadeau, Alain Giorgetti, and Abdallah Ben Othman. Praspel: A Specification Language for Contract-Based Testing in PHP. In B. Wolff and F. Zaidi, editors, ICTSS'11, 23th IFIP Int. Conf. on Testing Software and Systems.

[EDGB12] Ivan Enderlin, Frédéric Dadeau, Alain Giorgetti, and Fabrice Bouquet. Grammar-Based Testing using Realistic Domains in PHP. In A-MOST'12, 8th Workshop on Advances in Model Based Testing, joint to the ICST'12 IEEE Int. Conf. on Software Testing, Verification and Validation.

[EGB13] Ivan Enderlin, Alain Giorgetti, and Fabrice Bouquet. A Constraint Solver for PHP Arrays. In CSTVA'13, 5th Workshop on Constraints in Software Testing, Verification and Analysis, joint to the ICST'13 IEEE Int. Conf. on Software Testing, Verification and Validation.

I. Enderlin, F. Bouquet, F. Dadeau, A. Giorgetti
Email: ivan.enderlin@femto-st.fr
Web: http://hoa-project.net/s/praspel

femto-st SCIENCES & DISC TECHNOLOGIES
INSTITUT FEMTO-ST, DÉPARTEMENT DISC
16 ROUTE DE GRAY
25030 BESANÇON CEDEX - www.femto-st.fr

A MDE platform for modeling and validation of Secure Information Systems

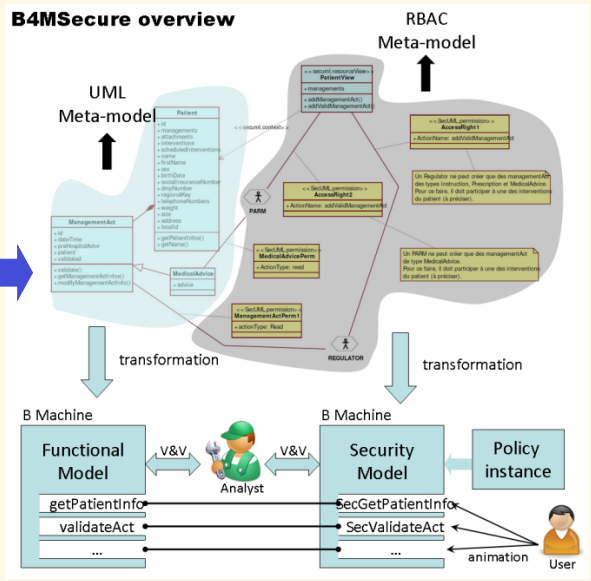
Akram Idani, Yves Ledru, Mohamed-Amine Labiadh
UJF-Grenoble 1 / Grenoble-INP / UPMF-Grenoble 2 / CNRS, LIG UMR 5217,
F-38041, Grenoble, France
{Akram.Idani, Yves.Ledru, Mohamed-Amine.Labiadh}@imag.fr

Access Control in Information Systems : Separation of Concerns

Today's organizations are extremely dependent on their Information Systems (IS). This is why corruption, loss or confidentiality breaking of their data can have serious consequences. Despite this fact, in most existing IS, functional and security requirements are mixed in the application code. It is, therefore, difficult to understand these systems and modify them in order to maintain, evolve and correct the security policy.

In order to master complexity of systems, the MDE paradigm advocates for a separation of concerns and the use of models throughout the development process. Then, Information Systems security is a domain where the potential of the MDE approach is highly useful. Indeed, modelling separately functional and security models allows to better understand, validate and maintain these models.

Although it is useful to analyse and validate both models in isolation, which is addressed by several works, interactions between these models must also be taken into account. Such interactions result from the fact that constraints expressed in the security model also refer to information of the functional model. Hence, evolutions of the functional state will influence the security behaviour. Conversely, security constraints can impact the functional behaviour. The B4MSecure platform allows, on the one hand, this separation of concerns, and on the other hand, the investigation of links between both functional and security models.

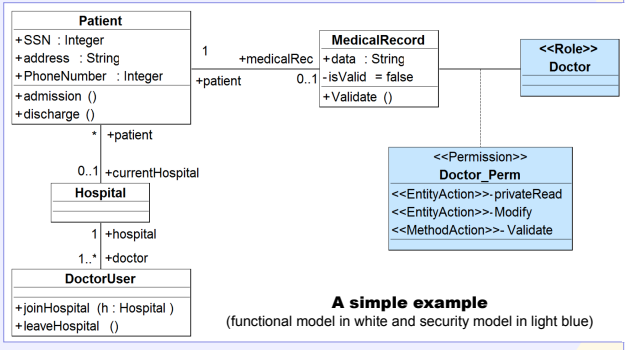


The tool

B4MSecure (http://b4msecure.forge.imag.fr) is an Eclipse platform dedicated to formally reason about functional UML models enhanced by an access control policy which follows the RBAC model. It is the result of the work done by the VASCO/LIG team in the national ANR project Selkis.

The platform acts on three steps :

1. Graphical modeling using the Topcased tool of a functional UML class diagram
2. Graphical modeling of an access control policy using a UML profile for RBAC (Role Based Access Control) and which is inspired by SecureUML
3. Translation of both models into B specifications in order to formally reason about them.



A simple example
(functional model in white and security model in light blue)

MACHINE
Functional_Model
SETS
MEDICALRECORD, PATIENT, ...
VARIABLES
MedicalRecord, Patient, isValid, ...
INVARIANT
MedicalRecord $\subseteq$ MEDICALRECORD $\wedge$
isValid $\in$ MedicalRecord $\rightarrow$ **bool**
...
INITIALISATION
MedicalRecord := $\emptyset$
...

$setData(mr, dd) \triangleq$
PRE
$mr \in MedicalRecord \ \wedge \ dd \in TheData$
$\wedge \ PatientMedicalRecordRel(mr) \in \mathbf{dom}(PatientHospitalRel)$
THEN
$isValid(mr) := FALSE \ \ ||$
$data(mr) := dd$
END;

$secure\_setData(mr, data) \triangleq$
PRE
$mr \in MedicalRecord \wedge data \in TheData$
THEN
SELECT
$MedicalRecord\_setData \in isPermitted[currentRole]$
THEN
$setData(mr, data)$
END
END;

How to take into account the following constaint in both models?
« In order to modify a medical record, the doctor must be employed by the current hospital of the patient »

```
context Doctor_Perm::Modify inv :
    session.user.isTypeOf(DoctorUser) implies
    session.user.Hospital = self.medicalRecord.Patient.currentHospital
```

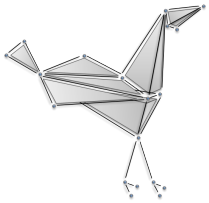
Cette recherche est soutenue par le projet SELKIS ANR-08-SEGI-018 et par l'ARC6 de la région Rhône-Alpes.

Computing Activity in Space

Martin Potier (LACL/U-PEC) | PhD with A. Spicher, O. Michel
martin.potier@u-pec.fr - http://www.lacl.fr/~mpotier
http://mgs.spatial-computing.org

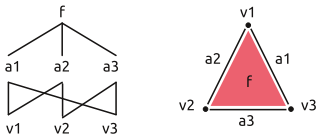
Context: MGS Concepts

MGS is a domain specific programming language dedicated to the modeling and the simulation of **Dynamical Systems with a Dynamical Structure**. It relies on topological concepts.

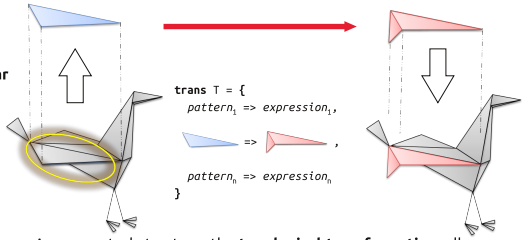


MGS' unique data structure is the **topological collection.**

Topological collections correspond to **Abstract Cellular Complex**, a concept borrowed from algebraic topology.



Cells of an ACC are bound by an incidence relationship.



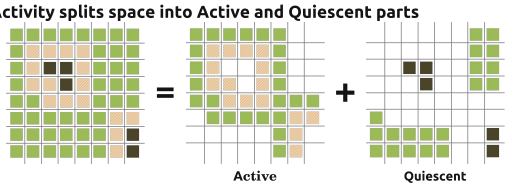
A new control structure, the **topological transformation**, allows the definition of case based functions matching the sub-parts of a topological collection which are replaced with an expression.

With MGS, computation is seen as a **topological rewriting**.

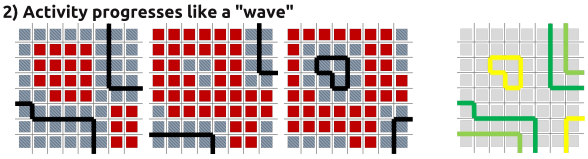
Activity in Space

Definition: **Activity** is a *measure of event* occurences or state changes in the simulation of a dynamical system.

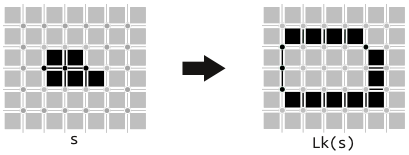
1) Activity splits space into Active and Quiescent parts



2) Activity progresses like a "wave"



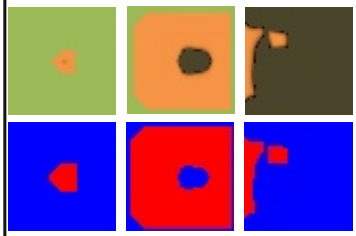
We illustrate the computation of activity with an example of a dynamical system: the spread of forest fire (1). The diffusion of the forest fire front gives a good clue on how activity progresses during a simulation (2).

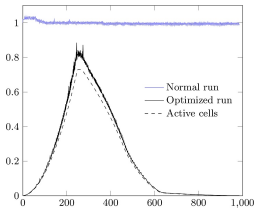


We use the **link** operator of combinatorial topology to *automatically* determine the next active area.

The advantages of this approach are twofold: first, activity opens the door to a *more efficient* pattern matching mechanism by focusing on the active part; second, it allows a better understanding of the dynamics of a model by identifying higher level structures and tracking them in simulations. In future work, we will consider the *reification of active objects* as first-order values that can be directly used in system specifications.

Application on Forest Fire



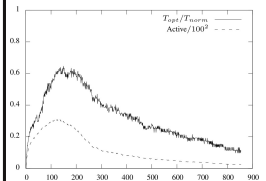


On the top, fire is in orange, forest in green and ashes are in brown. On the bottom, active cells are in red, and quiescent cells are in blue. Activity enlightens the fire front, an important emergent structure of fire spread models.

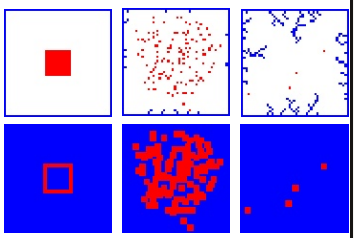
On the graph, we compare a normal run time (blue) to an optimized run time (black) at each iteration of the simu-lation. The speed-up is op-timum

Based on the work of *I. Karafyllidis and A. Thanailakis* in Ecological Modeling (1997) and implemented in MGS.

Application on Diffusion-Limited Aggregation



The graph represents the time of the optimized run on the time of the normal run at each iteration of the simu-lation. In this interaction process, the active part over-approximates the population of moving particles, thus the speed-up is not op-timum. However, the simulation time remains improved linearly with the size of the active part.



On the top, moving particles are in red and static particles are in blue. On the bottom, active cells are in red, and quiescent cells are in blue. In this cellular automaton, activity automatically recovers the po-pulation of moving particles.

Based on the work of *T. A. Witten and L. M. Sanders* in Phys. Rev. Lett. (1981) and implemented in MGS.

Modif : Automating data migration for the reuse of legacy tools

Paola Vallejo, Jean-Philippe Babau, Mickäel Kerboeuf
{vallejoco, babau, kerboeuf}@univ-brest.fr
Lab-STICC, MOCS Team, UBO

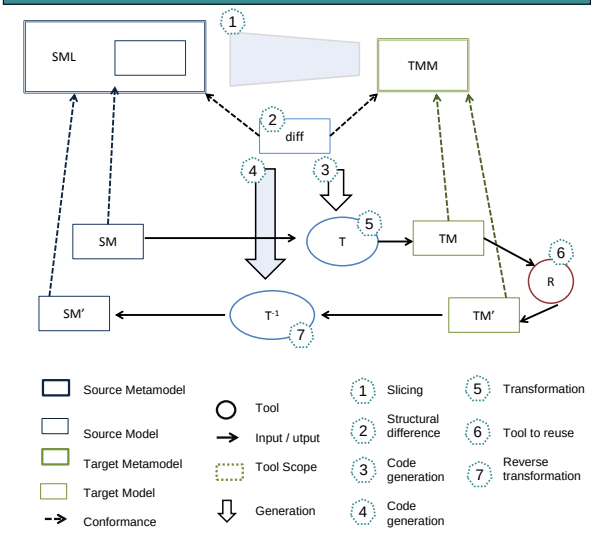
Needs

- Put data conforming to a specific metamodel under the scope of a targeted tool
- Make easier the data transformation from a source metamodel to a target metamodel and the reverse transformation to translate the data into their original context
- Lower the cost of obtaining the complete tool support for a standard modeling language
- Metamodel evolution and model co-evolution in order to promote the reusability of legacy tools

Proposal

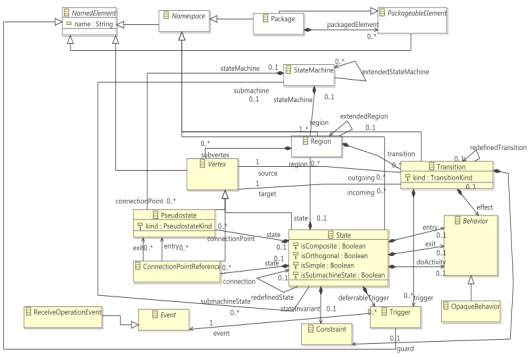
- **Modif :** a specific transformation language for metamodel refactoring
- Steps:
  - **Slicing:** to obtain a smaller effective metamodel
    - **Basic operators:** remove, rename, change multiplicity, change abstract, change containment
    - **Macro operators:** hide, flatten
  - **Contextualization:** to put back outcome data of a legacy tool into the specific context where the tool is reused
    - Keys

Modif Workflow



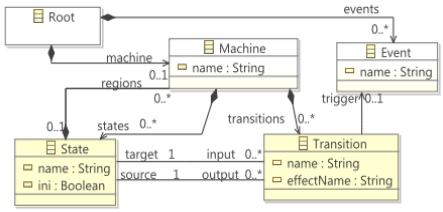
Slicing example

Source Metamodel



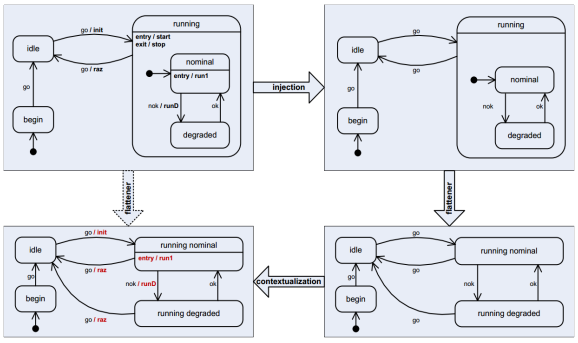
OPERATORS:

- Rename:
  - Package to Root
  - Region to Machine
  - ReceiveOperationEvent to Event
- Remove:
  - OpaqueBehavior
  - Constraint
  - ConectionPointReference
- Flatten/hide
  - Vertex
  - NamedElement
  - Event
  - PackageableElement
- Hide
  - Namespace
  - StateMachine
  - Trigger
  - Behavior



Target Metamodel

Contextualization example



ORQA: Modeling Energy and Quality of Service within AUTOSAR Models

Borjan Tchakaloff[1,2], Sébastien Saudrais[1], and Jean-Philippe Babau[2]

[1] Embedded Systems team, ESTACA, Laval, France
[2] MOCS, CACS team, Lab-STICC, UEB, UBO, Brest, France
borjan.tchakaloff@estaca.fr

1 Introduction

The Electric Vehicle (EV) has now reached an industrial maturity. Though several models are available, its energy capacity remains low, limiting its purpose to a day-to-day usage (about a hundred kilometers).

At the present time, most of the existing EVs provide no complex energy management: they only limit the vehicle speed when the battery is getting low without any concern about the driver's intentions or destination. The driver takes benefit of a *full service* while there is a certain amount of energy left and a *restricted service* otherwise. In *full service*, the driver is not restrained in any way. In the *restricted service*, the vehicle is limited to reduced speeds and some devices are restrained to keep the energy consumption at its lowest. This policy neglects the driver's preferences and is not optimal for a given trip. It would be necessary to anticipate the imposed reduced speed to reach a destination. Also, the driver may want to express preferences and priorities on devices usage and speed limitation policy. The challenge is then to provide an adaptable and acceptable solution between the two extremes.

To perform an efficient and adequate global energy management, a full control of configuration of all the consuming devices (respecting the driver's preferences) is required. This control is possible through the software embedded in the network of control units composing the vehicle information system. Each consuming device such as the lamps, the air-conditioning and so on is controlled by a dedicated control unit. The control software has to be integrated respecting the AUTOSAR (AUTomotive Open System ARchitecture) standard constraints.

The AUTOSAR consortium gathers automotive manufacturers and equipment makers in need for a common methodology. It aims at easing re-usability of Embedded Systems (ES) and contributing to a common basis, thus allowing easier project management and content sharing. The AUTOSAR methodology is based on models and relies on the software components paradigm. It permits designers to split ES modeling on different levels, from a system view down to the implementation code. But AUTOSAR models are architecture oriented and does not offer extra-functional properties support.

The energy consumption is an extra-functional property and, as such, is not taken into account in AUTOSAR models. In order to estimate the vehicle consumption, the consumption knowledge of every consuming device is required. Estimating what the vehicle should consume in certain conditions (the route type, the maximum vehicle velocity, the turned-on devices, etc.) allows to optimize the driving strategy. As the driver wants to reach his destination, the system should be able to offer at least one viable solution.

We propose ORQA (mOdeling eneRgy and Quality of service in Autosar), a framework to model the vehicle devices consumption and user-oriented Quality of Service. They are used to fulfill the driver's expectation: to reach a destination using as much as possible all the devices. To assure the driver of his success, the vehicle energy consumption has to be predicted for the available routes and the best route proposed to the driver. These predictions rely on the energy consumption knowledge of the whole vehicle, that is on both compulsory devices (the engine, the lamps, ...) and non-critical devices (the air-conditioning system, the heater, the auto-radio, ...). The framework presented in this paper takes into account both types of devices. Furthermore, an

on-line control of devices usage has to be performed to ensure the strategy realization with regards to the driver preferences. This results in an embedded energy manager that manages the vehicle devices taking into account their energy consumption and the driver's goal.

2 ORQA overview

The Orqa process is realized in two phases.

The first phase is performed at design stage. It is the creation of the energy model. The designer defines the power requirements of the vehicle devices using specific models. To help the designer, the framework offers a library of pre-defined models for the engine and each devices, the lamps, the Climate Control unit, the entertainment system devices and smaller consumers gathered in one constant power requirement.

The second phase is performed at run-time and concerns devices usage. It operates as follow:

1. The user chooses the target destination point and selects a consumption policy. The departure point is defined as the current vehicle position. The consumption policy is used to select the driving strategy and device usage policies.
2. Several routes can exist between the departure point and the destination. A GPS unit typically offers three types of routes: the fastest, the shortest and a trade-off between duration and distance. Orqa retrieves at most one of each route type proposed by the GPS unit.
3. Velocity coefficients (reducing maximal allowed speeds to lower speeds) are applied to the set of available routes, creating a matrix of route trips to evaluate. An evaluation results in the energy consumption of mandatory devices and functions, and the duration of a route with a specific velocity coefficient.
4. A driving strategy is selected from the available routes. The routes are first filtered by a maximum consumed energy, depending on the energy left in the battery and on a safety margin. Then, scores are assigned to the remaining routes depending on the user objective and the best match is selected. The user is then informed of the chosen strategy.
5. During the trip, the consuming optional devices are controlled by an embedded Energy Manager (a special component communicating with the device drivers). Following Autosar, architecture is static and cannot be modified at run-time. So the manager influence the devices behavior by the mean of brokers attached to their driving components.

The two phases process is based on a set of specific energy oriented models.

3 Conclusion

We propose a framework to ensure an Electric Vehicle driver that he will reach his destination point. This framework is realized by embedding a system-wide energy manager. The manager is based on a pre-computed model of energy consumption for all devices, a set of user preferences and different levels of Quality of Service given for each optional device.

From these models, the framework searches for available routes and computes for each of them duration and consumption for both the nominal driving and for reduced velocities. The framework relies on an energy model defining the consuming devices embedded in the vehicle to compute the global consumption. This model is introduced aside of the current embedded systems modeling done in Autosar. It is used to generate an enhanced Autosar model, which does not break the compatibility with existing tool-chains.

We are working on using the approach to optimize the different proposed levels and velocity coefficients, by exploring different cases in different configurations. The final goal is to define optimal strategies to embed in the vehicle. Also, we are working on bringing the overall accessories Quality of Service sooner in the process, in the route trips evaluation. Route trips can have close scores evaluating their fitness to the user consumption-policy. But some let more available energy for the accessories, making the accessories QoS increases. The idea is to take into consideration the accessories QoS sooner, so it can be optimized along the route path.

CIAO : modèle de composants et framework OSGi pour des applications télécoms adaptables dynamiquement

Areski Flissi[1] and Gilles Vanwormhoudt[1,2]

[1] LIFL/CNRS - Université Lille 1 (UMR 8022)
[2] Institut TELECOM
59655 Villeneuve d'Ascq cedex - France
{Areski.Flissi, Gilles.Vanwormhoudt}@lifl.fr

Abstract. Nous présentons CIAO (Components for sIp ApplicatiOns), un modèle de composants hiérarchique et dynamique, spécifique au domaine des services télécoms, ainsi que son implémentation sous la forme d'un framework d'exécution au dessus de la plate-forme OSGi. L'originalité de CIAO est qu'il permet de concevoir des applications télécoms avancées adaptables dynamiquement.

1 Introduction

Avec l'évolution rapide des réseaux IP d'une part, l'apparition et l'adoption de nouveaux protocoles tel SIP d'autre part, le domaine des services télécoms, et en particulier le développement d'applications avancées prenant en compte divers aspects tels la présence, la mobilité, la localisation, etc., nécessite aujourd'hui de nouvelles méthodologies et techniques inspirées du génie logiciel. Nous avons proposé dans [2], un modèle de programmation basé sur les notions d'acteurs, sessions et rôles afin de répondre aux différents challenges posés par la conception d'applications faisant intervenir plusieurs entités distribuées impliquées dans des interactions complexes. Dans [1], nous avons développé un outillage IDM, ainsi qu'un langage dédié basé sur ce modèle. Nous proposons ici de nous intéresser à l'évolution de ces applications, après le déploiement ou durant l'exécution, afin d'adapter leur comportement ou d'ajouter de nouvelles fonctionnalités dynamiquement. Dans ce poster, nous présentons CIAO (Components for sIp ApplicatiOns), un modèle de composant spécifique au domaine des applications télécoms répondant à cet objectif, ainsi que son implémentation sous la forme d'un framework d'exécution, reposant sur la plate-forme OSGi.

2 Modèle de composants CIAO

Nous avons défini un modèle de composants hiérarchique et dynamique, spécifique aux applications télécoms basées sur SIP. Celui-ci fait intervenir trois principaux types de composant qui sont:
– Un composant acteur nommé `Actor` qui représente une entité distribuée communiquant, via un flux de messages SIP, avec d'autres entités. Un acteur peut participer à différentes sessions (de même type ou non) avec d'autres acteurs. Un composant acteur est un composite encapsulant des composants `SessionPart` représentant chaque participation d'un acteur à une session. Le composite acteur a en charge l'aiguillage du flux de messages SIP au niveau des `SessionParts` et la coordination de ceux-ci (*i.e.* création d'une nouvelle instance, transmission d'un message à un composant `SessionPart` existant, etc.).
– Un composant `SessionPart` représente une participation d'un acteur à une session[3] donnée. `SessionPart` encapsule l'ensemble des comportements de l'acteur (*i.e.* ces rôles) au sein d'une session. Ainsi, un composant `SessionPart` est un composite contenant un ensemble de composants `Role`, dont il gère le cycle de vie.
– Un composant `Role`, qui est la brique de base du modèle contenant tout ou partie du comportement d'acteur relativement à une session. Ce comportement consiste à réaliser la logique métier en fonction de l'état et des messages SIP échangés.

[3] Nous entendons ici par session, un échange de messages persistent entre plusieurs acteurs

La particularité de ce modèle est la gestion dynamique des composants, basée sur les sessions SIP réelles. Les différents composants sont créés/détruits dynamiquement en fonction des flux de messages SIP, de l'état de l'acteur, des sessions, etc. Adapter dynamiquement une application en cours d'exécution consiste donc simplement à instancier ou détruire de nouveaux types de composants. Au sein d'un acteur, les composants interagissent pour router et traiter les messages SIP échangés, grâce à des ports spécifiques. Il existe également des moyens de coordination entre les composites et les sous-composants afin de supporter des comportements complexes impliquant plusieurs sessions d'un acteur ou plusieurs rôles d'une session.

3 CIAO : un framework au dessus de la plate-forme OSGi

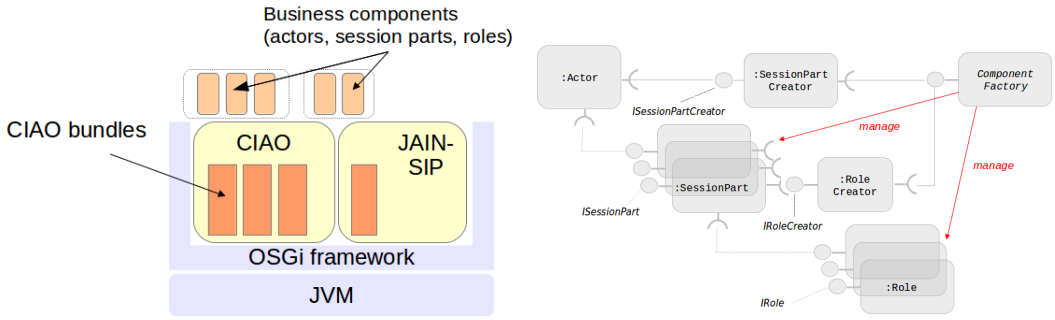


Fig. 1. Architecture et implémentation des composants CIAO

Le framework CIAO se base sur OSGi et l'architecture JAIN-SIP, comme illustré sur la figure 1 (partie gauche). Le choix d'OSGi a en partie été guidé par les fonctionnalités dynamiques proposées par cette spécification et plus particulièrement les composants *Declarative Service* (DS) présents depuis la version 4 d'OSGi, qui permettent de s'abstraire des problèmes de gestion dynamique des dépendances entre services requis et fournis des composants. Les composants CIAO ont donc été concrètement implémentés avec des composants DS. La partie droite de la figure 1 montre en détail l'assemblage des composants CIAO. La gestion de la dynamicité, donc du cycle de vie des composants `SessionPart` et `Role` a été rendu possible graĉe à l'utilisation du service *ComponentFactory* offert par la plate-forme OSGi, par les composants `SessionPartCreator` et `RoleCreator`. CIAO permet l'ajout, le retrait et le remplacement des comportements d'acteurs relativement aux sessions. Ces adaptations peuvent être appliquées sans interruption de l'application, grâce à une gestion de vie du cycle des composants qui tient compte des sessions en cours.

References

1. Areski Flissi and Gilles Vanwormhoudt. Programmation orientée domaine pour les services télécoms : concepts, DSL et outillage. In *Conférence en Ingénierie du Logiciel (CIEL 2012)*, pages 1–6, Rennes, France, 2012.
2. Gilles Vanwormhoudt and Areski Flissi. Session-based Role Programming for the Design of Advanced Telephony Applications. In *Proceedings of Distributed Applications and Interoperable Systems (DAIS'11)*, pages 77–91, 2011.